M.S. THESIS

THE OPTIMIZATION OF CONTEXT-BASED BINARY ARITHMETIC CODING IN AVS2.0

BY
CUI JING
FEBRUARY 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

공학석사 학위논문

AVS2.의 Context-based Binary Arithmetic Coding 최적화

The Optimization of Context-based Binary Arithmetic Coding in AVS2.0

2016 년 2 월

서울대학교 대학원

전기 정보 공학부

최 정

The Optimization of Context-based Binary Arithmetic Coding in AVS2.0

지도 교수 채 수 익

이 논문을 공학석사 학위논문으로 제출함
2016 년 2 월

서울대학교 대학원
전기 정보 공학부
최 정

최정의 공학석사 학위논문을 인준함
2016 년 2 월

위 원 장 최 기 영 (인)

부위원장 채 수 익 (인)

위 원 이 혁 재 (인)

초론

HEVC(High Efficiency Video Coding)는 지난 제너레이션 표준 H.264/AVC 보다 코딩 효율성을 향상시키기를 위해서 국제 표준 조직과(International Standard Organization) 국제 전기 통신 연합(International Telecommunication Union)에 의해 공동으로 개발된 것이다. 중국 작업 그룹인 AVS(Audio and Video coding standard)가 이미 비슷한 노력을 바쳤다. 그들이 많이 창의적인 코딩 도구를 운용한 첫 제너레이션 AVS1 의 압축 퍼포먼스를 높이도록 최신의 코딩 표준(AVS2 or AVS2.0)을 개발했다.

AVS2.0 중에 엔트로피 코딩 도구로 사용된 상황 기반 2 진법 계산 코딩(CBAC)은 전체적 코딩 표준 중에서 중요한 역하를 했다. HEVC 에서 채용된 상황 기반 조정의 2 진법 계산 코딩(CABAC)과 비슷하게 이 두 코딩은 다 승수 자유 방법을 채용해서 계산 코딩을 현실하게 된다. 그런데 각 코딩마다 각자의 특정한 알고리즘을 통해 곱셈 문제를 처리한 것이다. 본지는 AVS2.0 중의 CBAC 에 대한 더 깊이 이해와 더 좋은 퍼포먼스 개선의 목적으로 3 가지 측면의 일을 한다.

첫째, 우리가 한 비교 제도를 다자인을 해서 AVS2.0 플랫폼 중의 CBAC 와 CABAC 를 비교했다. 다른 실행 세부 사항을 고려하여 HEVC 중의 CABAC 알고리즘을 AVS2.0 에 이식한다.예를 들면, 상황 기반 초기치가 다르다. 실험 결과는 CBAC 가 더 좋은 코딩 퍼포먼스를 달성한다고 알려진다.

그 다음에 CBAC 알고리즘을 최적화시키기를 위해서 몇 가지 아이디어를 제안하게 됐다. 코딩 퍼포먼스 향상시키기의 목적으로 근사 오차 보상(approximation error compensation)과 확률 추정 최적화(probability estimation)를 도입했다. 두 코딩은 다른 앵커보다 다 부호화효율 향상 결과를 얻게 됐다. 다른 한편으로는 코딩 시간을 줄이기를 위하여 레테 추정 모델(rate estimation model)도 제안하게 된다. 부호율-변형 최적화 과정(Rate-Distortion Optimization process)의 부호율-변형 대가 계산(Rate-distortion cost calculation)을 지지하도록 리얼 CBAC 알고리즘(real CBAC algorithm) 레테 추정(rate estimation)을 사용했다.

마지막으로 2 진법 계산 디코더(decoder) 실행 세부 사항을 서술했다. AVS2.0 중의 상황 기반 2 진법 계산 디코딩(CBAD)이 너무 많이 데이터 종속성과 계산 부담을 도입하기 때문에 2 개 혹은 2 개 이상의 bin 평행 디코딩인 처리량(CBAD)을 디자인을 하기가 어렵다. 2 진법 계산 디코딩의 one-bin 제도도 여기서 디자인을 하게 됐다. 현재까지 AVS 의 CBAD 기존 디자인이 없다. 우리가 우리의 다자인을 관련된 HEVC 의 연구와 비교하여 설득력이 강한 결과를 얻었다.

주요어: 오디오 및 비디오 코딩 표준(AVS); AVS2.0;상황 기반 2 진법 계산 코딩(CBAC);상황 기반 조정의 2 진법 계산 코딩(CABAC);비교 제도; 근사 오차 보상; 확률 추정; 레테 추정;2 진법 계산 디코딩 건축

학번:2013-22510

Abstract

High Efficiency Video Coding (HEVC) was jointly developed by the International Standard Organization (ISO) and International Telecommunication Union (ITU) to improve the coding efficiency further compared with last generation standard H.264/AVC. The similar efforts have been devoted by the Audio and Video coding Standard (AVS) Workgroup of China. They developed the newest video coding standard (AVS2 or AVS2.0) in order to enhance the compression performance of the first generation AVS1 with many novel coding tools.

The Context-based Binary Arithmetic Coding (CBAC) as the entropy coding tool used in the AVS2.0 plays a vital role in the overall coding standard. Similar with Context-based Adaptive Binary Arithmetic Coding (CABAC) adopted by HEVC, both of them employ the multiplier-free method to realize the arithmetic coding procedure. However, each of them develops the respective specific algorithm to deal with multiplication problem. In this work, there are three aspects work we have done in order to understand CBAC in AVS2.0 better and try to explore more performance improvement.

Firstly, we design a comparison scheme to compare the CBAC and CABAC in the AVS2.0 platform. The CABAC algorithm in HEVC was transplanted into AVS2.0 with consideration about the different implementation detail. For example, the context initialization. The experiment result shows that the CBAC achieves better coding performance.

Then several ideas to optimize the CBAC algorithm in AVS2.0 were proposed. For coding performance improvement, the proposed approximation error compensation and probability estimation optimization were introduced. Both of these two coding tools obtain coding efficiency improvement compared with the anchor. In the other aspect, the rate estimation model was proposed to reduce the coding time. Using rate estimation instead of the real CBAC algorithm to support the Rate-distortion cost calculation in Rate-Distortion Optimization (RDO) process, can significantly save the coding time due to the computation complexity of CBAC in nature.

Lastly, the binary arithmetic decoder implementation detail was described. Since Context-based Binary Arithmetic Decoding (CBAD) in AVS2.0 introduces too much strong data dependence and computation burden, it is difficult to design a high throughput CBAD with 2 bins or more decoded in parallel. Currently, one-bin scheme of binary arithmetic decoder was designed in this work. Even through there is no previous design for CBAD of AVS up to now, we compare our design with other relative works for HEVC, and our design achieves a compelling experiment result.

Keywords: Audio and Video coding Standard (AVS), AVS2.0, Context-based Binary Arithmetic Coding (CBAC), Context-based Adaptive Binary Arithmetic Coding (CABAC), comparison scheme, approximation error compensation, probability estimation, rate estimation, Binary Arithmetic Decoder (BAD) Architecture.

Student number: 2013-22510

Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Research Background

Recent years, with the rapid development of the information technology, the demand for the multi-media, such as video media, is getting greater and greater. Mass data offered by the video carrier make the information storage and transmission more difficult to handle and it is necessary to explore the effective and efficient video compression technique, especially in the vast images data and real-time transmission with high definition requirement. The video compression and coding technique has been significantly enhanced since it merged in 1980s. The main procedure of video codec includes prediction for video images to obtain the residual data, transform and quantization for the residual data, entropy coding for the data after quantization, as well as the bit-stream collection finally. However, a reverse procedure is performed for the decoder part, and the reconstruction video sequence is achieved through bit-stream as input. The typical video codec structure can be described as Fig.1-1.

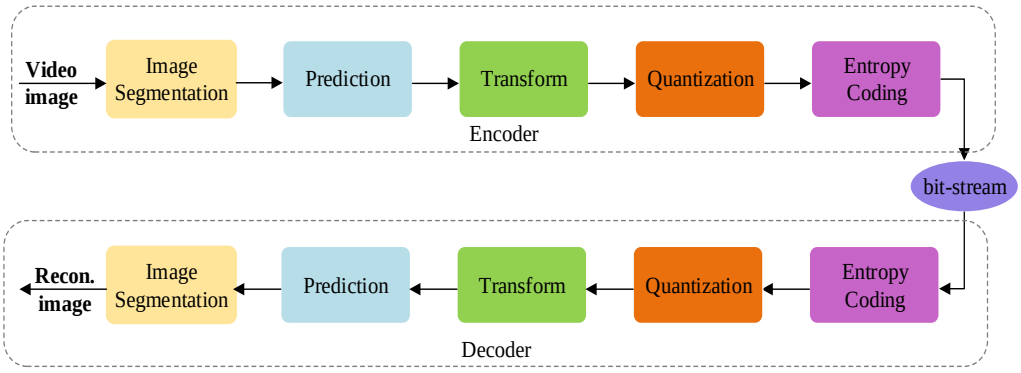

Figure 1- 1 The typical video codec block diagram

Many efforts have been made by the video expects from the International Telecom Union (ITU) , Video Coding Expert Group (VCEG), International Standard Organization (ISO) and Moving Picture Expert Group (MPEG) in the past several decades and consequently there are considerable development in the video compression standards. H.261 is the first generation motion image compression standard developed by the ITU[1] followed by the H.263 standard proposal[2] which was developed for the low bit rate video coding at the Nov. 1995. H.263 was aimed to the low bit rate compression for the high quality motion image and used to support the application with bit rate less than 64kbits/s. In the following several years, ITU proposed couple improved vision based on H.263. IMEG family [3] including MPEG-1, MPEG-2, MPEG-4, MPEG-7, and MPEG-21 have been developed by the ISO. Until at the beginning of the 21-st century, H.264/AVC [4] introduced by the ITU and ISO brought about 50% performance improvement compared with MPEG-2 and has been popular in the industrial application. At the same time, another video standard, named AVS[5] developed by the Audio Video coding Standard (AVS) Workgroup in China. The coding complexity was deduced compared with the H.264/AVC with a comparable coding efficiency. Along with the new high definition and ultra-high definition video requirements, High Efficiency Video Coding (HEVC) [6] were proposed and finished the final draft in 2013 by the Joint Collaborative Team on Video Coding (JCT-VC) which is the cooperative team including ITU VCEG and ISO MPEG. This standard has been designed aim to save over 50% [7] bit rate to get the comparable quality, albeit at

higher computational costs. Correspondingly, AVS workgroup has spared more efforts to make second generation video codec orientated to higher coding efficiency referred as AVS2.0 [8]. Specifically, the video technique can be represented as the Fig.1-2 according to the development in the past 30 years.

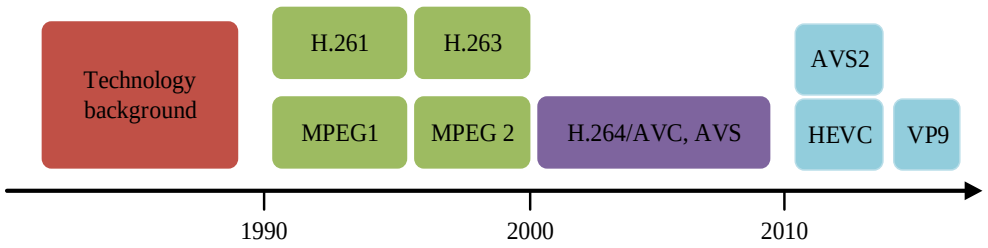

Figure 1- 2 The development of video codec standard

1.2 Key Techniques in AVS2.0

Similar with other mainstream video coding standard, the overall coding framework of AVS2.0 can be shown in Fig.1-3.

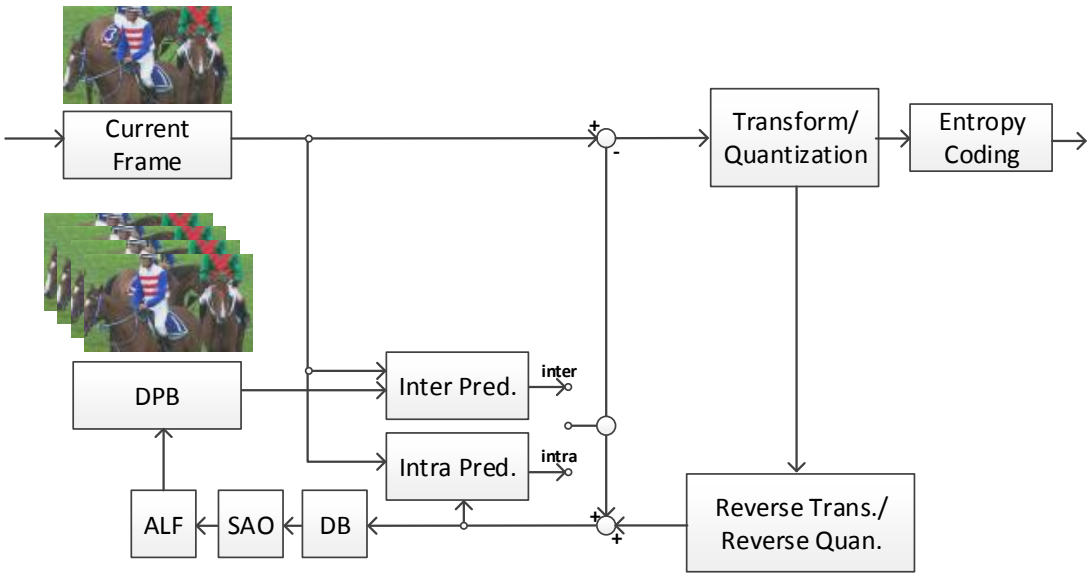

Figure 1- 3 The coding block diagram of AVS2.0

However, the specific techniques introduced into AVS2.0 standard includes Intra prediction, Inter prediction, Transform & Quantization, Entropy coder, Sample adaptive offset, and Adaptive loop filter [9]. With the similar algorithm structure of HEVC, AVS2.0 has the competitive coding efficiency but more simplified algorithms for each mode to deal with video image. Although the coding procedure of AVS2.0 shares the similar structure of HEVC, AVS2.0 pays more attention on some special application scene, such as surveillance video, real-time video meeting, etc. Specifically, for each part, including Intra prediction, Inter prediction, Transform/Quantization, Entropy coding and Loop filter, technique baseline and performance improvement in BD-rate saving (%) in AVS2.0 are presented in Table 1-1.

Table 1- 1 Key techniques used in AVS2.0

Type	**Technique baseline**			**Coding gain**
Image structure	Hierarchical reference frame	B picture used as reference	Forward multiple hypothesis prediction picture	8% ~ 13%
Block structure	Quad-tree based coding unit partitions	Non-square intra prediction	Non-square inter prediction Non square transform	15% ~ 20%
Intra prediction	33 directional prediction modes	1/32 sub pixel prediction		5% ~ 10%
Inter prediction	Forward multiple hypothesis prediction, special prediction mode and motion vector prediction	Progressive motion vector coding	DCT like interpolation filter	7% ~ 12%
Transform	Multiple size and highly normalized integer transform	Secondary transform		3%
Entropy coding	Two level scan coding			5%
Loop filter	Deblock filter	Sample adaptive offset	Adaptive loop filter	8%

Then we will briefly introduce the key feature of each technique adopted in AVS2.0.

A. Block Structure

The block partition is more adaptive compared with AVS1.0 by using quad-tree structure. The 64*64 is the largest coding unit (LCU) and then it is partitioned into smaller coding unit (CU) until reaching the minimum coding unit limitation size 8*8. Through this partition mode, then coding tree (CTU) structure is obtained. Fig.1-4 gives the quad-tree partition structure.

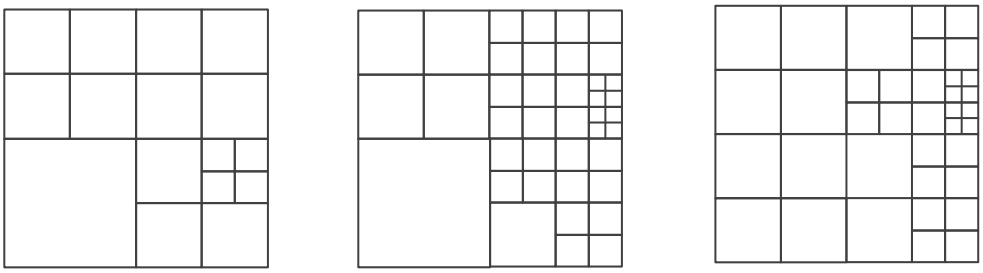

Figure 1- 4 The quad-tree partition structure in AVS2.0

Each CU then can be divided into some prediction unit (PU), PU is the basic unit for intra and inter-picture prediction. For intra prediction, there are four type PUs among which N*N PU is used for 8*8 CU only and 2N*0.5N/0.5N*2N are introduced in CU size 32*32 and 16*16. Eight types PU are used in inter prediction, including 2N*2N、N*N、N*2N、2N*N、2N*nU、2N*nD、nL*2N、nR*2N. The maximum PU size is decided by the current CU and minimum PU is 4*4. The transform unit (TU) is another coding block which is used for the transform and quantization operations. TU is also decided by the current CU size without consideration the PU size anyway, 64*64 and 4*4 are the maximum and minimum TU size, respectively. Fig.1-5 is the prediction coding unit partition structure.

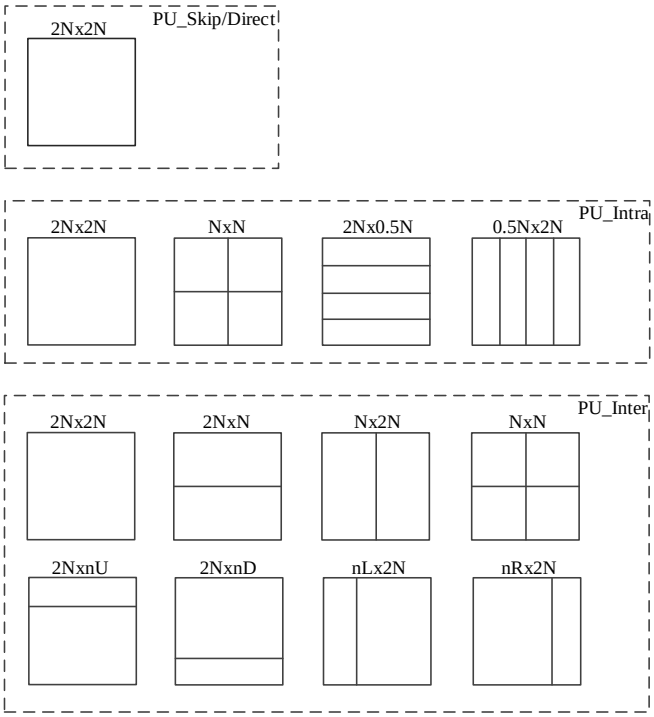

Figure 1- 5 The prediction unit structure in AVS2.0

B. Intra Prediction

Intra prediction is employed to remove the spatial redundancy within picture. Multi-direction intra-picture prediction is used in AVS2.0 and as described in *A* section, except for four partitions, the Short Distance Intra Prediction (SDIP) [10] is used for intra prediction on 32*32 and 16*16 CU. Fig.1-6 shows 33 modes including DC, Plane, Bilinear and 30 Angle modes for luma component.

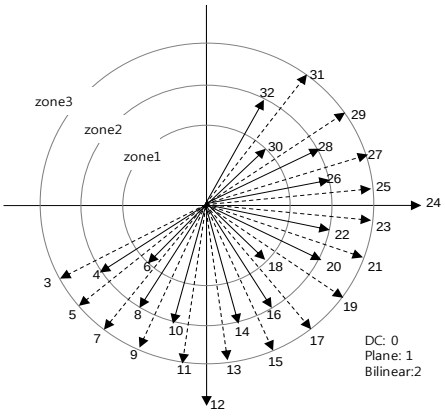

Figure 1- 6 Intra prediction direction in AVS2.0

C. Inter Prediction

Inter prediction is employed to remove the spatial redundancy between picture. AVS2.0 uses 8 inter prediction modes as described in *A* section, and 3 frame types: P frame, B frame, and F frame. F frame is developed based on the P frame with bi-forward inter prediction. In inter prediction, there are specific techniques patented by AVS2.0 developer group, including Dual Hypothesis Prediction (DHP) [11], Directional Multi-Hypothesis Prediction (DMH) [12], Progressive Motion Vector Resolution (PMVR) [13], etc.

D. Transform & Quantization

In AVS2.0, the two-level transform coding to deal with residual data. Firstly, using Wavelet Transform and then DCT transform as the TU size is divided into 32*32. In DCT transform, 4*4 ~ 32*32 TU size are supported and Non-Square Quad-tree Transform (NSQT) is used to handle non-square TU. In order to reduce the information redundancy, the residual data will be performed a second DCT transform [14].

In addition, Rate Distortion Optimization Quantization (RDOQ) is another technique adopted by the AVS2.0 in the rate distortion optimization process. RDOQ makes the compromise between the computation complexity and the coding efficiency. To reduce the complexity to decide mode, only is the mode within the one coding unit decided, the RDOQ is used for the coefficients quantization in the best mode in AVS2.0.

E. Entropy Coding

The entropy coding in AVS2.0 is only context-based binary arithmetic coding (CBAC), which is different from AVS1.0 where CBAC and variable length coding technique are performed as entropy coders. In CBAC, two-level transform coefficient coding scheme acts as the well-designed entropy coding strategy. The two-level scheme [15] employs the similar concept of sub-block based partition as in HEVC and applies this scheme to the *(Level, Run)* coefficients pair of large blocks. In this scheme, the sub-block size is set to a fixed value with 4×4 and named as one coefficient group (CG) in the following text.

Entropy coding plays a vital role in the entire coding structure as the Fig.3 illustrates. It locates in the last step of the encoder and the first step of decoder which determines the bin-to-bit compression ratio which is relative the coding performance. Entropy coding, especially CBAC is the study center in this research topic, and more detail will be shown in the following several chapters.

F. Loop Filter

To reduce the visual flaw caused by the video coding algorithm, there are three methods used in AVS2.0 including Deblocking Filter (DF), Sample Adaptive Offset (SAO) [16], and Adaptive Loop Filter (ALF) [17] to address the visual problem for the reconstructive picture.

Even through a significant compression efficiency has been achieved by AVS2.0 based on the above techniques compared with AVS1.0, the improvement in each

technique perspective can be explored to make it better enough to comparable with other popular video coding standard, such H.264/AVC, HEVC etc. However, in order to escape the copyright and patents own by other standards, the techniques employed in AVS tend to be more complexity and simpler in the algorithm implementation. Thus, the study on the AVS2.0 is full of challenge in the algorithm design and schedule implementation practically.

1.3 Research Contents

In AVS2.0, context-based binary arithmetic coding (CBAC) [18] is the only entropy coding method introduced into current standard. In this thesis, there are three topics we focus on the entropy coder CBAC in AVS2.0. Firstly, we compare performance between two entropy coder with different algorithm, which are CBAC and context-based adaptive binary arithmetic coding (CABAC) that is used in H.264/AVC and HEVC. Secondly, we propose some ideas about the CBAC performance enhancement and then introduce the fast rate estimation model for the AVS2.0 in the rate distortion optimization (RDO) mode decision process. Lastly, we implemented Binary Arithmetic Decoder with throughput of one-bin per cycle, which is main bottle-neck of implementation of CBAC Decoder with high throughput. More detail will be shown in the following several subclasses.

1.3.1 Performance Comparison of CBAC

We propose a fair scheme to compare the CBAC with Context-based Adaptive Binary Arithmetic Coding (CABAC) [19] in HEVC, as Fig.1-7 shows, we implant

CABAC logic that is designed for HEVC into RD10.1, which is one of latest versions of reference software of AVS2.0. The coding efficiency of AVS2.0 using two entropy coders can be evaluated by bitstream 0 and bitstream1, which are from the result of encoding the given video sequence.

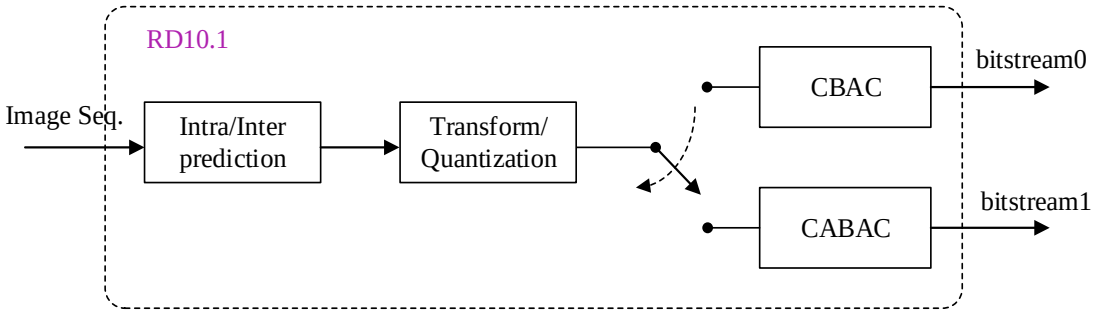

Figure 1- 7 scheme for comparison between two entropy coders.

Through comparison of these two entropy coders, we can obtain the knowledge about entropy coding compression performance. Our evaluation experiments show that CBAC algorithm tend to be more efficient than CABAC with about 0.4% BD-rate saving when we use the CABAC algorithm of HEVC directly to encode the same video sequences.

1.3.2 CBAC Performance Improvement

With understand of the reason of coding efficiency improvement, we explore more in CBAC algorithm in AVS2.0. Most of algorithms in Codec are usually used to implement without using multiplier operation to reduce Complexity of Computation. In the process of updating variables, which is used for Arithmetic Coding such as range and context probability, multiplier operations are replaced with other operations similarly. Look-up table is used in CABAC in HEVC for the purpose of

this. While the logarithm addition and shift operations is used in CBAC. But, introduction of operation of logarithm domain necessarily accompany the process to convert data between real domain and logarithmic domain, which requires additional computational complexity. So CBAC uses two approximation equations to minimize overhead by domain conversion. For that reason, it is likely to increase coding performance if we can reduce approximation error at the sake of minimal increase of computational complexity.

Therefore, we present compensation tables to minimize the error by approximation equations within the CBAC engine by introducing adjusted factors when the approximation equations are used in domain conversion.

Adaptive probability estimation [20] [21] is another topic in CBAC which is a powerful optimization to indict how to map the symbol statistical behavior. Based on the fact that probability estimation in CBAC is also performed in the logarithm domain with probability in certain bits resolution, we explore the probability estimation scheme with the perfect bit resolution and well-designed update process.

In addition, rate estimation is introduced into AVS2.0 in order to save the overall encoding time. Different from AVS2.0 software reference, we use the proposed rate estimation table to support the rate distortion cost in the Rate-Distortion Optimization (RDO). Though the proposed rate estimation model, the encoding time can be reduced about 1% without considerable performance degradation.

1.3.3 Implementation of Binary Arithmetic Decoder in CBAC

Through the above two chapters in the algorithm study, we understand the software implementation detail better. Based on this understand, the hardware-oriented architecture for binary arithmetic decoder is described in this chapter. Considering the total CBAC decoder will cost more time to arrange reasonable context models, only Binary Arithmetic Decoder (BAD) with one bin scheme is designed in this chapter, but we give the proposed context update module architecture. For the BAD, there are three important loops needed to update after one bin is decoded, which includes range update loop, offset update loop and bits read. Correspondingly, we design three modules to realize the update: range update module, offset update module, bits read module. Since few previous work is focus on the CBAC decoder in AVS2.0, we compare our work with the available CBAC decoder design in AVS1, and the competitive result can be achieved based on our BAD architecture.

1.4 Organization

Chapter 2 describes the entropy coding CBAC in AVS2.0 and how it works the arithmetic engine. Also, the two-level transform coefficients coding is given in detail. In Chapter 3, the coding efficiency of CBAC and CABAC of HEVC are compared based on the software platform of AVS2.0 RD10.1. We proposed a quite fair comparison scheme with consideration of initial context variables, binarization, adaptive probability estimation model, etc. In Chapter 4, we propose some idea to improve coding efficiency in CBAC such as error compensation, new probability estimation scheme and introduction of rate estimation table. Then, we describe how

to implement binary arithmetic decoder in CBAC in Chapter 5. In the last Chapter, the research conclusion about this thesis and further research orientation are posted.

Chapter 2 Entropy Coder CBAC in AVS2.0

2.1 Introduction of Entropy Coding

Context-based Adaptive Binary Arithmetic Coding (CABAC) is a method of entropy coding first introduced in H.264/AVC, and it is also adopted in the newest standard - High Efficiency Video Coding (HEVC). Similar with the method used in above standards, another kind of entropy coding approach – Context-based Binary Arithmetic Coding (CBAC) is introduced in a Chinese video standard – Audio and Video coding standard (simplified as AVS) by the Audio Video coding standard of Workgroup of China. However, the strong data dependence and serious operations in nature make entropy coding more complicate to parallelize and improve the throughput. Thus in the design of standard of entropy coding for H.264/AVC, HEVC, and AVS, the balance of coding efficiency and throughput should be considered.

Specifically, all the current entropy coding engines are based on the arithmetic coding [22] [23]. Arithmetic coding is different from other coding methods because we know the exact relationship between the coded symbols and the actual bits that are written to a file. It codes one symbol once, and a real-valued number of bits is assigned to each symbol. The code value *v* of a compressed data sequence is the real number with fractional digits that equals to the sequence's symbol. We can convert sequence. This construction create a convenient mapping between infinite sequences of symbols from a D-symbol alphabet and real numbers in the interval [0, 1), where any data sequence can be represented by a real number, and vice-versa. This kind of code value

presentation can be used in any coding system, and it makes a universal method to represent large amounts of information of a set of symbols used for coding, such as binary, decimal, etc. By analyzing the distribution of the code value it produced, we can evaluate the efficiency of any compression method. According to Shannon's information theory, we can know that, if a coding method is optimal, then the code values cumulative distribution has to be a straight line from point (0, 0) to the point (1,1). When it is applied into video coding, it is attached with context information of each symbol. Therefore, entropy coding is the kind of lossless compression approach which can use the statistical probabilities of source information, e.g. video or image carriers, so that a string of bits can be used to represent the symbols is logarithmically proportional to the corresponding probability of each symbol. When compressing a string of symbols, the symbol which occurs in a large frequency can be represented by few bits, while the other symbols with less frequent emergence, represented with a longer bit string. According to the Shannon's information theory, the probability of a symbol represented in bit 0 or 1 is *p*, the optimal average code length for one symbol is $-\log_2 p$.

In the general videoing coding standard, the classical codec framework is represented as Fig.1. And the entropy coding is performed in the last step of the overall video coding after the video signal has been parsed to series of syntax elements. Correspondingly, it is in the first stage of the video decoding procedure in each standard.

2.2 CBAC Overview

The CABAC algorithm is firstly introduced within the joint H.264/AVC standard of ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG). CABAC was used as one of two alternative methods of entropy coding in H.264/AVC, and introduced as the only method in HEVC.

Similarly, the entropy coding in AVS 1.0 jizhun file includes two schemes, C2DLVC and CBAC, which not only adopted 2-dimension (run, level) coding scheme used in MPEG-2, but also absorbed the context-based adaptive binary arithmetic coding strategy used in H.264/AVC. In C2DVLC, the VLC multiple tables achieved by training in off-line. It is not able to capture the local statistical distributions in nature and a symbol with a probability which is greater than 0.5 cannot be coded efficiently considering the nature limit to 1 bit/symbol in VLC codes. However, the arithmetic coding can challenge this restriction with a higher coding efficiency.

Therefore, in this section, the CBAC algorithms and separated key technique are represented systemically from the AVS1.0 to AVS2.0. The general procedure for the CBAC includes binarization, context derivation and selection, and arithmetic coding engine. And these compounds illustrated in Fig.2-1. The binarization process is aimed to translate the values of the non-binary syntax elements into binary and it is defined as the bin string generation process. The context derivation and selection process is related to the probability modeling process, in which the each bin can be mapped into a specific context to estimate the probability of each regular bin. Finally, the binary arithmetic coding process is adopted to compress the bins into bits according to the context

information and probability distribution. There are two kinds of the arithmetic coding paths according to the probability value for each bin, including the regular path and bypass path.

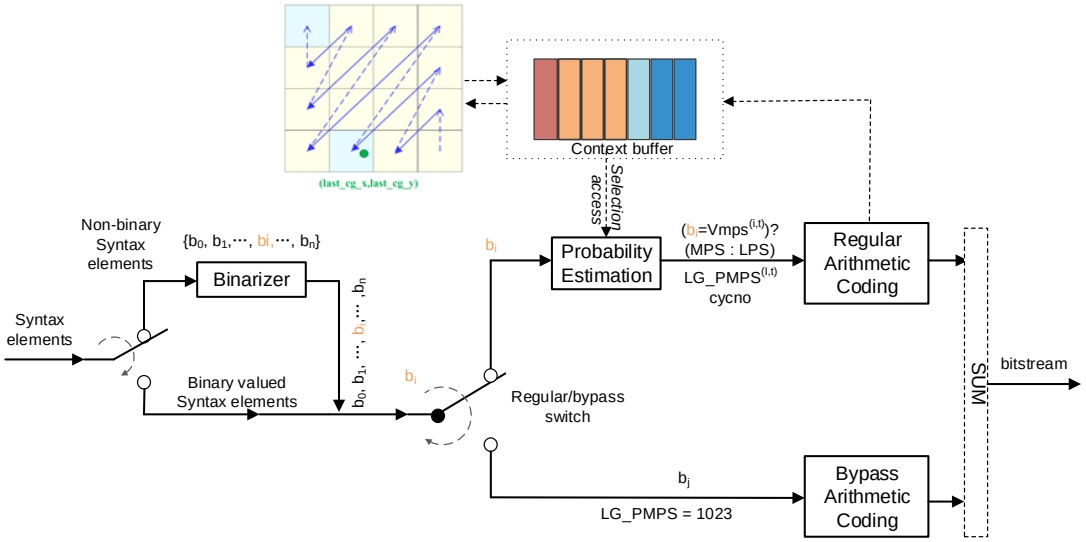

Figure 2- 1 The general block diagram of CBAC in AVS2.0

2.2.1 Binarization and Generation of Bin String

Binarization process is aimed to uniquely map process of all possible values of a syntax element onto a set of bin string. For the non-binary valued symbols, e.g. Level and Run, they should be performed the binarization process as the values of this kind of syntax elements tend to be typically in a large range in a DCT block. When this value is coded directly by the m-ary (m>2) arithmetic code, it will have a high computation complexity. Moreover, the source with typically large alphabet size often suffers from “context dilution” effect when the high-order conditional probabilities have to be estimated on a relatively small set of coding samples. In addition, the context modeling for the sub syntax element level provides more accurate probability estimation than that in the

syntax elements level, and the alphabet of the encoder is decreased.

There are several methods of binarization adopted in video coding standard. All of these methods, including Unary, Truncated Unary, *k-th* order Exp-Golomb (EGk), and Fixed length are introduced to reduce the alphabet size of syntax elements to encode.The binarization methods for syntax elements which are applied into the CBAC of AVS2.0 represented as the following [24]:

(1) Unary coding is used to binarize the symbol into a bin string with length N+1, including the first N bins with value 1 and the last bin is 0.

(2) Truncated Unary scheme is defined based on the largest possible value *maxVal* of the syntax element. Before *maxVal*, the binarization value is the same as the Unary, and when the value is equal to *maxVal*, all the bins in the bin string are set to 0 and the total bins are the same as that of the *maxVal* -1.

(3) Marking bit is defined as the bin value is the same as the value of the syntax element.

(4) The *k-th* order Exp-Golomb coding with *k* ranged from 0, 1, 2, 3, has a general construction, which consists of a prefix and suffix. For the given codeNum *N* and the specific order k, the code word consists of *l* zeros followed by one 1 and suffix of $N\text{-}2^k(2^l\text{-}1)$, and the *l* is defined as following:

$$l = \min\{0, \lceil \log_2((N+1)/2^{k+1}) + 1/2 \rceil\} \qquad (2\text{-}1)$$

However, except for the above several schemes, for most syntax elements in CBAC, the binarization process is defined based on the type of the syntax element.

2.2.2 Context Modeling and Probability Estimation

Context Modeling Process, shown in Fig.2-1, consists of three sub steps: context model derivation, context model selection andcontext model access. The context modeling process is referred as the probability selection process. In the regular binary arithmetic coding process, where the probability model is decided by the fixed modelbased on the type of the syntax elements and the bin position or the bin index in the binarized representation of the syntax elements. Another kind of context (probability model) is adaptively chosen from the two or more than two probability models according to the side information, such as the special neighbors(Left, Above block), components (Luma, Chroma), depth and size of the CTU, PU, TU as well as the position of within one TU. The adaptive case is generally adopted into the observed bins with high frequency while the fixed model is usually applied for the less frequently occurred bins. Thus the modeling process can be benefited from the balance of the choice cost and context learning complexity with the estimated accuracy.

Similar with probability models in CABAC adopted in H.264/AVC and HEVC, the CBAC probability updating model is based on the adaptive probability model as well, in which the parameters of the probability model make a promising contribution to the map the statistical variations of the source bins which is performed bin-by-bin basis as the sub symbol. This is the probability estimation process. The derivation of the CBAC probability updating process is applied for the infinitely independent identical distribution (IID) [25] of the binary source. If the probability of the symbol “1” is p, and the probability of the symbol “0” is q. And the adjusting parameter N is defined to

adjust the updating speed. Then p_k and q_k are defined as the estimated probability of the symbol "1" and "0" after the *k-th* iteration. And then we can achieve the probability after *(k+1)-th* iteration as the following equation 2-1:

$$\begin{cases} p_{k+1} = \dfrac{N \cdot p_k}{N+1} & (if \ \text{"0"} \ occurs) \\ q_{k+1} = \dfrac{N \cdot q_k}{N+1} & (if \ \text{"1"} \ occurs) \end{cases} \tag{2-2}$$

According to the relationship between *p* and *q*, i.e. $p_k = 1 - q_k$, the equation (2-2) can be changed as the following equation (2-3):

$$\begin{cases} p_{k+1} = \dfrac{N \cdot p_k}{N+1} & (if \ \text{"0"} \ occurs) \\ p_{k+1} = \dfrac{1}{N+1} + \dfrac{N \cdot p_k}{N+1} & (if \ \text{"1"} \ occurs) \end{cases} \tag{2-3}$$

According to the above equations, the expectation and variance of the p_{k+1} are proved to converge to a constant value which is dependent on N. Therefore, if we use the p_{MPS} and p_{LPS} as the probabilities of the MPS and LPS symbol, thus the probability change can be obtained based on the equation (2-2), as the following equation (2-4):

$$\begin{cases} p_{MPSnew} = \alpha \cdot p_{MPSold} & (if \ \ LPS \ \ occurs) \\ p_{LPSnew} = \alpha \cdot p_{LPSold} & (if \ \ MPS \ \ occurs) \end{cases} \tag{2-4}$$

Here $\alpha \leftarrow \dfrac{N}{N+1}$. That is to say, the larger the *N* is, the α is smaller, the slower the estimation converges, the variance is smaller, thus the probability estimation is more accurate.

However, in H.264/AVC and HEVC, the probability estimation model is based on the assumption that the estimated probabilities of each context model can be represented by a sufficiently limited numbers of representative values. For the CABAC engine,

there are 64 limited representative probability values p, which is ranged from 0.01875 to 0.5, including. The estimation model can be derived from the recursive equation of the LPS symbol as the following (2-5):

$$p_\delta = \alpha \cdot p_{\delta-1} \quad (\delta = 1, 2, 3, \ldots \tag{2-5}$$

$$\text{With } \alpha = (\frac{0.01875}{0.5})^{1/63} \text{ and } p_0 = 0.5$$

The scaling factor $\alpha \approx 0.95$ and the probability state is set as 64, in which the compromise of the speed and estimation accuracy. Each probability p_δ is addressed according to the probability state.

As to the practical implementation procedure, In CABAC of H.264/AVC and HEVC, the probability state updating process is based on the 64-state Finite State Machine (FSM). In this process, the state transfer process is performed to index a pre-defined state table, where the state is the index, and state is also the key variable for each context.

Similarly, In AVS1 and AVS2.0, the context modeling adopts the same probability estimation model to model the information source and performing probability updating process for each context. However, since CBAC and CABAC apply different schemes to perform the entropy coding, the probability modeling process is experienced various procedure, especially in the term of practical implementation. In AVS, the state of probability estimation model is based on the logarithm value of probability, which is scaled into 10-bit resolution domain (0 ~ 1024) in theory. Therefore, the probability model is based on the probability and logarithm value of the probability of MPS symbol. The scaled probability *LgPmps* can be described as equation (2-6):

$$LgPmps = 1024 \times |\log_2(p_{mps})| \qquad (2\text{-}6)$$

Here, p_{mps} is the MPS probability. Thus for each probability including MPS and LPS are indicted in the scaled probability LgPmps when it implemented in CBAC. The statistics of the coded syntax elements are utilized to update the probability models, which is related to context models of regular bins. Therefore, more specific explanation of the transition rules for updating the state indices will be shown in binary arithmetic coding, and contexts design derivation sections.

2.2.3 Binary Arithmetic Coding Engine

The basic principle of arithmetic coding is introduced in [22], which is based on the recursive interval subdivision of the interval width R. Each binary symbol of the information source which is represented by a bin string, associated with a specific context model, which keeps update during the coding process in order to adaptively estimate the probability. Therefore, the variables for BAC is bin value, slice type, and the context model for each bin. And BAC is a recursive process of the coding interval (range, offset, low) subdivision, updating, and renormalization operations as Fig. 2-2.

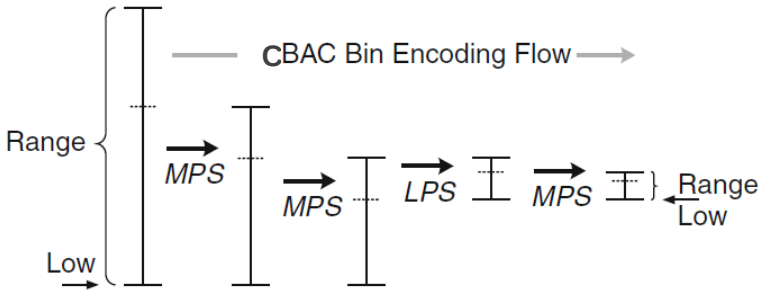

Figure 2- 2 Subdivision and decision procedure of BAC

A given interval initially which can be represented as the lower bound *L* and range *R* is

subdivided into two sub-ranges according to an relative estimation of the probability p_{lps} valued from 0 to 0.5, not including, of the Least Probability Symbol (LPS).Thus another part can be described as p_{mps} and subrange R_{mps} of Most Probability Symbol (MPS). One of the sub-range can be denoted as the following equation:

$$R_{mps} = p_{mps} \times R \qquad (2\text{-}7)$$

Which is associated with the MPS symbol and corresponding interval of the range LPS $R_{lps} = R - R_{mps}$, which is related to the MPS with a probability $p_{mps} = 1 - p_{lps}$. According to the binary value to be encoded, the relative LPS or MPS range will be chosen as the new interval for the next iteration.

Based on the above description, the subdivision is performed via the multiplication, but multiplication operation is proven with high computation complexity and calculation cost both in software and hardware. The practical implementation method has been focus on the multiplication-free operations, such as the look-up table approach which is used in H.264/AVC and HEVC, where a well-developed table is pre-designed, the sun-range can be obtained from the look-up table operation. Thus the multiplication operation is eliminated. However, the CBAC in AVS2.0 is based on a novel algorithm which is based on the domain conversion between logarithm and original domain. By this method, the multiplication operation can be substituted by the logarithm adder operation in logarithm domain. More detail about the two methods to reduce the multiplication complexity will be represented in the following sections.

In CABAC, the BAC is performed on the look-up table to realize the range subdivision

and applies for the FSM to deal with the state transition for the context and probability updating. However, the procedure in CBAC in AVS2.0 experience a various scheme. The process is an iterative one which consists of consecutive *MPS* symbols and one *LPS* symbol. 9-bit precision for range is kept during whole coding process. In the binary arithmetic coder of CBAC, we substitute the multiplication in (2-7) with addition by using logarithm domain instead of original domain. When a *MPS* happens, the renewal of range is given as

$$LgR_{mps} = LgR + Lgp_{mps} \tag{2-8}$$

where Lgx indicates the logarithm value of variable x and LgR_{mps} is the new range after encoding one *MPS*. For the case of encountering one *LPS*, we denote the two *MPS* range before and after encoding the *LPS* as R_1 and R_2 as shown in Fig. 2-3. Then, the range after the whole coding cycle in original domain should be

$$R_{lps} = R_1 - R_2 \tag{2-9}$$

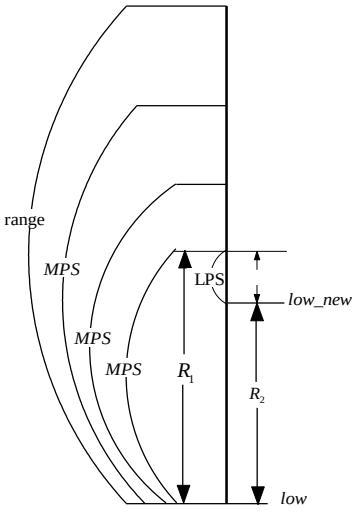

Figure 2- 3 One binary arithmetic coder cycle

And the new lower bound of current range equals to the addition of *low* and R_2. Since

R_1 and R_2 are both calculated on the logarithm domain, we have to get the value of R_1 and R_2 from LgR_1 and LgR_2, and then

$$R_1 = 2^{LgR_1} = 2^{-s_1+t_1} \approx 2^{-s_1} \times (1+t_1-\Delta_1) \qquad (2\text{-}10)$$

and

$$R_2 = 2^{LgR_2} = 2^{-s_2+t_2} \approx 2^{-s_2} \times (1+t_2-\Delta_2) \qquad (2\text{-}11)$$

Here, s_1 s_2 are the integer, and t_1 t_2 are the fraction part, which range from [0, 1). Δ_1 and Δ_2 are the approximation error adjust factor. From (2-10), (2-11), we can get the following, ignoring the approximation error Δ_1 and Δ_2:

$$R_{lps} = 2^{-s_1} \times t_3 \qquad (2\text{-}12)$$

and

$$t_3 \approx \begin{cases} t_1 - t_2 & if\ (s_2 == s_1) \\ (t_1 << 1) - t_2 & if\ (s_2 = s_1 - 1) \end{cases} \qquad (2\text{-}13)$$

After the new value of R_{lps} is obtained, the renewed lower bound is updated. Then the renormalization process is carried out to guarantee that the most significant bit of the updated *range* value is always '1'. Until now, one coding cycle is finished. After one bin is encoded by arithmetic coder, the estimated probability of the chosen context should also be updated. In order to prepare the relative parameters for the next iteration, the range in original domain should be exchanged into logarithm domain. Considering a fact that the approximation will stand when the variable x ranged into a small interval (0, 1) as following:

$$\ln(1+x) \approx x \quad (0 < x < 1) \qquad (2\text{-}14)$$

The integer part of the logarithm-based updated range R_{lps} s1 is 0, and the fraction part t_3 can be simplified with the above equation. Thus the R_{lps} in logarithm domain can be obtained and the range preparation for the next cycle is finished.

Actually, in CBAC, the probability of each context model is set to be 0.5 for both *MPS* and *LPS* at the start of coding initially. With the coding of some bins, the adaptive probability estimation of *MPS* on logarithm domain is performed. Based on the context modeling section described in section 2.2.2, the practical probability estimation is fulfilled using only additions/subtractions and shifts as in the following formulas:

$$\begin{cases} LgPmps \leftarrow LgPmps + Lgf & (if\ \ lps) \\ LgPmps \leftarrow LgPmps - (LgPmps >> cw) & (if\ \ mps) \end{cases} \quad (2\text{-}15)$$

Where *f* is equal to $(1\text{-}2^{-cw})$. Here, *cw* is the size of sliding widow to control the speed of probability adaptation. The smaller *cw* is, the faster the probability adaptation will be. In the practical implementation process, the *cw* is adaptive according the *cycno* parameter, which is adopted to record the iteration of calling the CBAC engine.

2.3 Two-level Scan Coding CBAC in AVS2.0

Different from AVS1, AVS2.0 supports larger transform blocks (e.g., 16×16 and 32×32). In the early stage of AVS2.0 standardization process, the CBAC design for AVS2.0 is inherited from that in AVS1 by a straightforward extension. However, CBAC was primarily designed for 8×8 transform blocks while the non-zero coefficients may be sparser in larger transform blocks. Therefore, to further improve the coding efficiency

and throughput issue in hardware implementation, AVS2.0 CBAC employs a two-level coefficient coding scheme [15].

Generally, the iteration of CBAC in AVS is slice, which means that all the binary arithmetic coding engine relative parameters will be initialized after finishing one slice. Only the syntax elements which are belong to the slice segment data, will be processed by the CBAC encoded. The coding structure in the slice illustrated as Fig.2-4, including slice header information, slice data information, the coding procedure in one LCU, and the slice end information. The syntax elements that are coded with CBAC in AVS2.0 include three categories: (1) context-based syntax elements, (2) bypass mode-based syntax elements, (3) stuffing bit-based syntax elements. For AVS, these context-based syntax elements describe the properties of the coding tree unit (CTU/LCU), coding unit (CU), prediction unit (PU), and transform unit (TU). For the CTU level, the related syntax elements are used to represent the block partition information of the CTU, the type including edge and band, and offsets for the sample adaptive offset (SAO), and adaptive loop filtering in loop filtering in CTU. For a CU, the syntax elements are related to describe whether the CU is intra prediction mode, or inter prediction mode, the PU type definition of B and F frame. For a PU, it includes the syntax elements which describe the intra prediction mode, and the motion data. For the TU level, the coding tree pattern, and residual data including transform coefficient, level and run information.

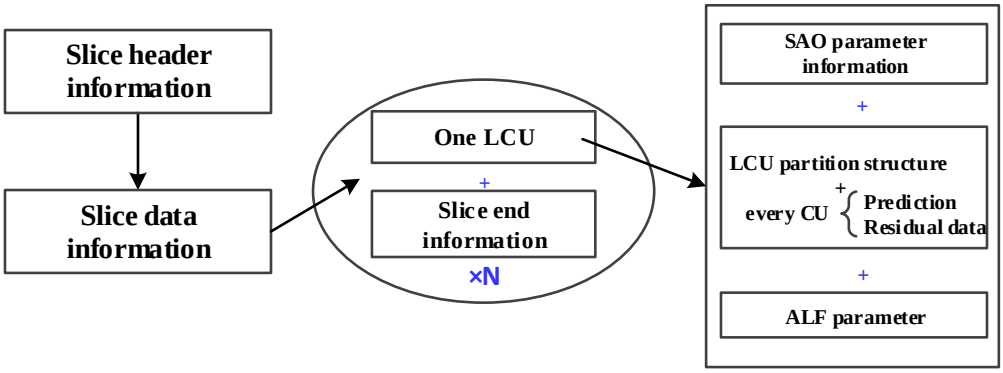

Figure 2- 4 The slice coding structure for the CBAC

However, entropy coding in AVS, which is the similar with CABAC in H.264/AVC and HEVC, provides a high coding efficiency, while its strong data dependence caused by the serious operations in its procedure put a big challenge on the throughput improvement. The throughput of CBAC is determined by the binary symbol that it can be performed per second. Moreover, the significant contribution is made by the syntax elements of transform coefficient data, which includes the residual of the prediction error.

The two-level scheme employs the similar concept of sub-block based partition as in HEVC [26] and applies it to the (*Level*, *Run*) coding to address the spatiality of large blocks. In this scheme, the sub-block size is set to a fixed value, i.e., 4×4. Such a sub-block is named one coefficient group (CG) in the following text. The CG level coding is firstly invoked, followed by the (*Level*, *Run*) coding within one CG which is similar to CBAC in AVS1.

2.3.1 Scan order

In CBAC for AVS2.0, the coefficient coding for a transform block (TB) is decoupled

into two levels, i.e., CG level coding and coefficient level coding. In both levels, the coding follows the reverse zig-zag scan order. Fig. 2-5 shows the zig-zag scan pattern in a TB with a different size, which is split into sub-blocks and the scan order of CGs is indicated by lines while the scan order within one CG is indicated as the line shows in Fig.2-6. The CG-based coding methods have two main advantages:

- Allowing for modular processing, that is, for harmonized sub-block based processing across all block sizes.
- With much lower implementation complexity compared to that of a scan for the entire TB, both in software implementations and hardware.

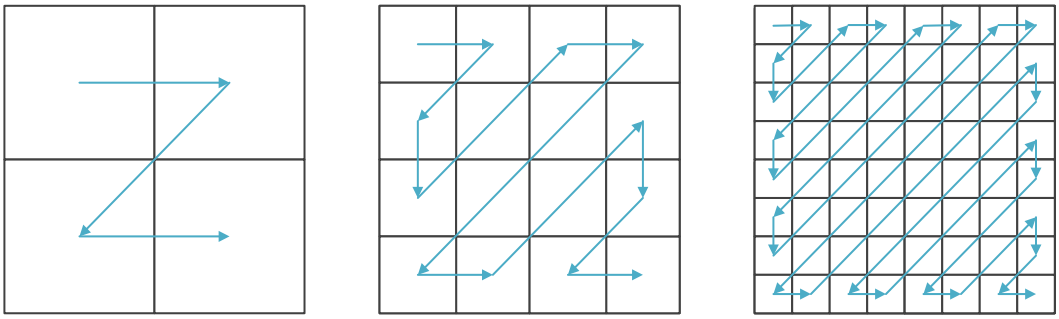

Figure 2- 5 Sub-block scan: each sub-block is a Coding Group (CG)

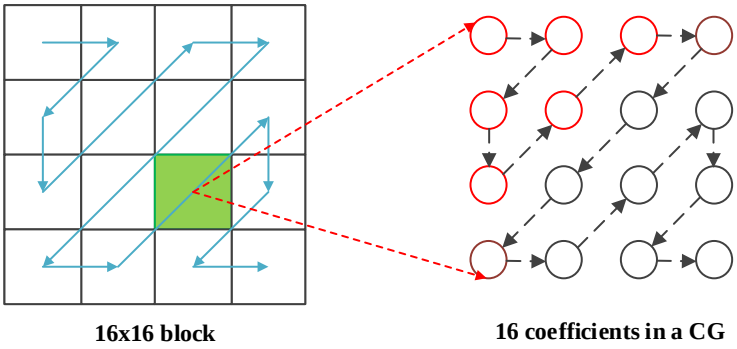

Figure 2- 6 4*4 Coefficients scan within a CG

2.3.2 First level coding

For the current coding block which is divided into multiple CGs as Fig.2-5 shows. The first level coding is performed among these CGs. At inter CG level, the position of the last CG is signaled, where the last CG is the CG that contains the last non-zero coefficient in the transform block in the scan order. Different ways are used to signal the position of the last CG which is dependent on the TB sizes. For an 8×8 block, a syntax element *LastCGPos* is coded, which is the scan position of the last CG. For larger TBs, such as 16×16 and 32×32 TBs, one flag *LastCG0flag* is firstly coded first to indicate whether the last CG is at position (0, 0). In the case that *lastCG0flag* is equal to one, two more syntax elements *LastCGX* and *LastCGY* are coded to signal the(x, y) coordinates of the last CG position. Note that, (*LastCGY*- 1) is coded instead of *LastCGY* when *LastCGX* is zero since *lastCG0flag* is equal to one.

The first level coding is performed by several syntax elements which indicate the information about the current CG in the entire TB. Thus the syntax elements for this level are explained by the *last_cg_pos*, *last_cg0_flag*, *last_cg_x*, *last_cg_y*, *last_cg_y_minus1* and *nonzero_cg_flag* and each description is presented in Table 2-1.

Table 2- 1 The syntax elements for the first level coding

Syntax elements	Description
last_cg_pos	denotes the position of the last CG block in the current TB
last_cg0_flag	indicates whether the last CG position is 0 or not in the TB (larger than 8x8)
last_cg_x	denotes the x coordinate of the current CG in the current TB
last_cg_y	denotes the y coordinate of the current CG in the current TB
last_cg_y_minus1	denote the y coordinate of the current CG in the current TB when the x coordinate is zero.
nonzero_cg_flag	signals whether the current CG includes non-zero coefficients

2.3.3 Second level coding

The second level coding indicates the coding of coefficients within one CG. Fig. 2-7 depicts the coding flow for one CG. Basically, it follows the principle of the CBAC design in AVS1. However, when one CG contains non-zero coefficients (i.e., the *nonzero_cg_flag* of the CG is equal to 1 or it is the last CG), the position of the last non-zero coefficient in the scan order in the CG is coded instead of coding the end of bit (EOB) flag after each *(Level, Run)* pair to signal a stop. Then, the (*Level*, *Run*) pairs are coded sequentially in the reverse scan order until the coding of all pairs are finished. Similar to the coding of (*Level*, *Run*) pairs in CBAC for AVS1, the *Level* is represented by its magnitude *absLevel* and the sign information.

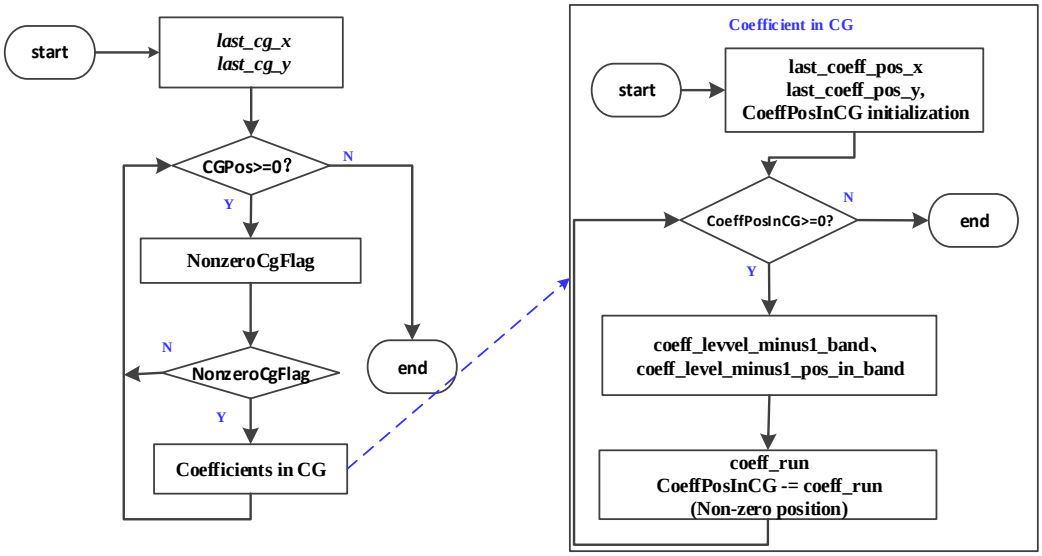

Figure 2- 7 Coding flow for the transform coefficients

It is observed that depending on whether the CG is the last CG, the distribution of the position of the last nonzero coefficient shows different exhibitions. As a result, two last coefficient position coding schemes are utilized accordingly. For the last CG, the position of the last non-zero coefficient in the CG is mostly random but has a general

tendency to be close to the top-left corner of the sub-block. The position is then directly coded in its (*x, y*)-coordinates relative to the top-left position of one CG, namely, *LastPosX* and *LastPosY*. For CGs which is not the last CG, the position of the last non-zero coefficient, if present, tends to be close to the bottom-right corner of the sub-block and is also highly correlated to the reverse scan order. It is therefore more efficient to code its reverse scan position within the CG rather than the (*x, y*)-coordinates, i.e., the position relative to the bottom-right position of one CG.The coding procedure in the second level is based on the coefficients in each CG and the coding order is the reverse order of the zig-zag scan. The syntax elements for this step can be defined as: *last_coeff_pos_x, last_coeff_pos_y, coeff_level_minus1_pos_in_band* and *coeff_run* and each description is presented in Table 2-2.

Table 2- 2 The syntax elements for the second level coding in one CG

Syntax elements	Description
Last_coeff_pos_x	Denote the x-coordinate of last non-zero coefficient in the nonzero CG.
***Last*_coeff_pos_y**	Denote the y-coordinate of last non-zero coefficient in the nonzero CG.
coeff_level_minus1	Denote the range of the coefficient level minus 1.
coeff_level_minus1_ pos_in_band	Denote the position of the coefficient level minus1 in the current level band.
coeff_run	denote the run value
coeff_sign	Indicate the coefficient is positive or not.

2.4 Summary

In this section, the detail about the entropy coding in AVS2.0 was presented in the above aspects.Then, the context-based binary arithmetic coding theory is analyzed, and the binarization, context modeling & probability estimation, and the binary arithmetic

coding engine are all summarized in detail. It is the complicated computations and strong data dependence that post more challenge on this topic about CBAC entropy coding.

Chapter 3 Performance Comparison in CBAC

The Context-based Adaptive Binary Arithmetic Coding is the typical entropy coding method used in current video coding standard, such as HEVC, H.264/AVC, AVS, etc. In order to understand the coding performance of tools contributed by the CBAC better, we proposed a comparison scheme to compare the entropy coder CBAC with CABAC based on the software reference RD platform of AVS2.0. In this chapter, we give the performance comparison though the proposed comparison scheme and to keep it fair, the adaptive context initialization is introduced when we transplant CABAC into reference s/w of AVS2.0 as CABAC used in reference s/w of HEVC adopts specific initial context variables for each context model. It is different form CBAC in AVS2.0 because the context variables of all context models in CBAC are initialized with the same value at the beginning of the new slice.

3.1 Differences between CBAC and CABAC

In H.264/AVC and HEVC, the CABAC is adopted as entropy coding technique, which is based on the Look-up table (LUT) operation to free multiplication. On the other hand, Logarithm Domain Addition (LDA) is used for CBAC in AVS2.0.

Generally, the Binary Arithmetic Coder (BAC) of current video standards mentioned above is consisted of three steps: (1) Binarization, (2) Context Modeling (Probability estimation and assignment), and (3) Arithmetic coding. The binarization is a procedure to map syntax elements with non-binary value into binary value with some elementary

schemes which are suitable model-probability distribution. The context modeling is a procedure to associate a probability model with different type of the syntax elements adaptively. The whole process of Selection of the probability model according to the syntax element type, bin index and the side information is referred as context modeling. In this process, the probability model parameters is adaptive in order to estimate the statistical feature of the source bins. Each binarized syntax element decided through rate distortion optimization (RDO) mode decision process is processed in BAC engine with matched context model for each bin the arithmetic coding will be finally performed based on the probability update and range subdivision.

Specifically, the CABAC algorithm is based on the LUT for range division and context update is realized through another two LUTs for MPS and LPS case. Each of LUT includes 64 states transiting according to the probability estimation model. And each context of syntax element includes 6-bit probability state indexing two context update LUTs and 1-bit value of MPS bin. However, the CBAC algorithm in AVS2.0 performs the entropy coding through the logarithm addition and shifting in order to eliminate the multiplication. The context model introduces 10-bit probability-based variable of MPS bin, 1-bit for the value of MPS bin and the 2-bit counter parameter marking sliding window size for the probability estimation. Different form that in CABAC where the sliding window size is fixed as about 19.69, the adaptive probability estimation model is introduced through 2-bit counter parameter in CBAC. Therefore, the differences between two entropy coders CBAC and CABAC can be summarized as the Table 3-1.

Table 3- 1 The differences between two entropy coders

		CBAC	CABAC
Binarization	Syntax elements	-	-
Context Modeling	Sliding window parameter	Adaptation (cycno, cwr)	Fixed
	Initial probability	Fixed	Adaptation
	Bit depth of probability	10 bits	8 bits
	Context model variables	Probability(10-bit scaled), valMps, cycno	Probability(LUTs), valMps
BAC	Method free multiplication	Logarithm addition	LUTs

3.2 Comparison of Two BAC Engines

In order to evaluate the coding efficiency of two BAC engines fairly, we design the specific comparison schemes for each engine. Firstly, we should transplant the CABAC engine into RD10.1, which is reference s/w of AVS2.0 and use it as the entropy coder to encode and decode the video sequence. Based on the differences presented in above Table 3-1, we can see that CABAC employs different method to realize the binary arithmetic coding, especially in the context modeling and arithmetic engine part. To compare fairly, then we need to consider how to make the two entropy coders in the same scheme to realize each step in their multiplication-free operations. Fig.3-1 gives the block diagram to compare two entropy coders CABAC and CBAB. However, in order to measure the coding efficiency of these two entropy coders, the comparison scheme [27] should be exactly matched the procedure in each standard. Thus, the significant issue needed to address is how to design the adaptive initialization value of probability for each context model of each syntax element in AVS2.0 when CABAC is used as entropy coder. In addition, there are several optimization methods used in the logarithm domain-based arithmetic CBAC. The adaptive probability estimation and

adaptive sliding window size are the techniques which can be used to improve the compression performance of arithmetic coding. However, in this evaluating scheme, what need to do is to keep the comparison fair and retain the original feature of each entropy coder used in respective video standard as much as possible.

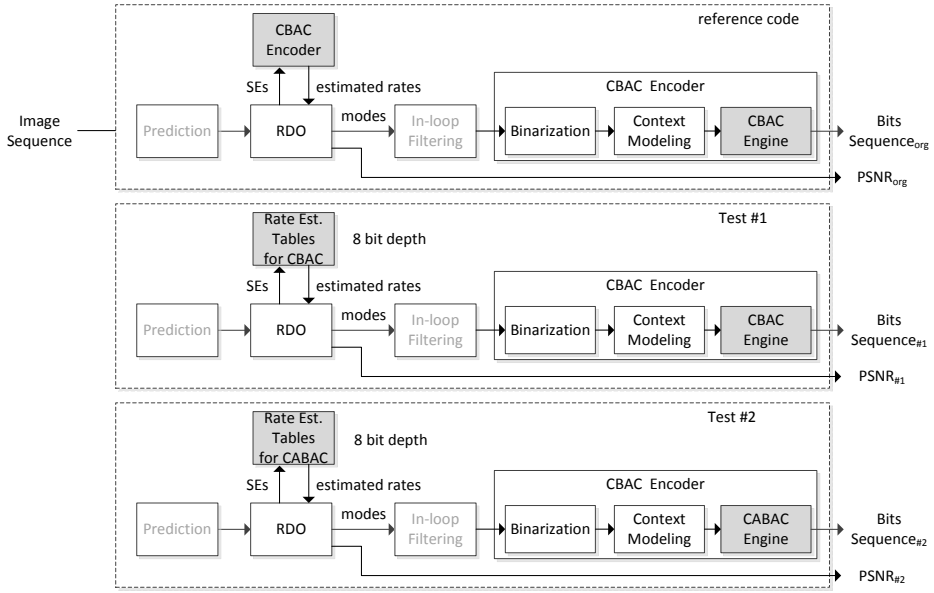

Figure 3- 1 The Block Diagram for Evaluating CBAC and CABAC Engines

3.2.1 Statistics and initialization of Context Models

The context model initialization process for each entropy coder holds some difference and we should reduce this distinction in proposed scheme. Specifically, the initial probability value for each context model of each syntax element is distinct in CABAC. It is one of conditions of CABAC which works for the only entropy coder in HEVC. Thus at the beginning of each slice, the context variable of probability in each context is assigned to the respective value. While the initialization of probability is performed as the assigning the same value 0.5 to each context model in CBAC in AVS2.0. Thus

the context initialization for the CBAC is pretty easy to perform as all the context models are set as the same initial value, including MPS symbol as 0, *LgPmps* rested as 1023, and the *cycno* parameter designed as the start iteration. However, according to source information in the nature video, the adaptive context in the different area even the same syntax elements tend to be set as the various initialized features. In addition, the different syntax elements should be assigned to the adaptive initial value at the beginning of the each slice. To achieve this goal, we should give the specific initial value of each context. Table 3-2 gives the syntax elements accessed to CBAC entropy coder. For some syntax elements, 2-D context buffer is used for the context updating to make scalability possible in future.

Table 3- 2 The context number of each syntax element in RD10.1

Syntax Elements	**Ctx num.**	**Syntax Elements**	**Ctx num.**
cuType_contexts	11+9	cbp_contexts	[3][4]
pdir_contexts	18	map_contexts	[8][17]
amp_contexts	2	last_contexts	[8][17]
b8_type_contexts	9	split_contexts	8
pdir_dhp_contexts	3	tu_contexts	3
b8_type_dhp_contexts	1	lastCG_contexts	30
b_dir_skip_contexts	4	sigCG_contexts	3
p_skip_mode_contexts	4	lastPos_contexts	56+16
wpm_contexts	3	saomergeflag_context	3
mvd_contexts	[3][10]	saomode_context	1
pmv_idx_contexts	[2][10]	saooffset_context	2
ref_no_contexts	6	m_cALFLCU_Enable_SCModel	[3][4]
delta_qp_contexts	4	brp_contexts	8
l_intra_mode_contexts	7	pdirMin_contexts	2
c_intra_mode_contexts	4		

The context initialization process is performed based on the fact that all the contexts in one slice will be initialized with the same variable. The initial procedure is described in Fig. 3-2, in which *biari_init_context_logac( )* function defines the initial context variables including *LgPmps*, *valMps* and *cycno*.

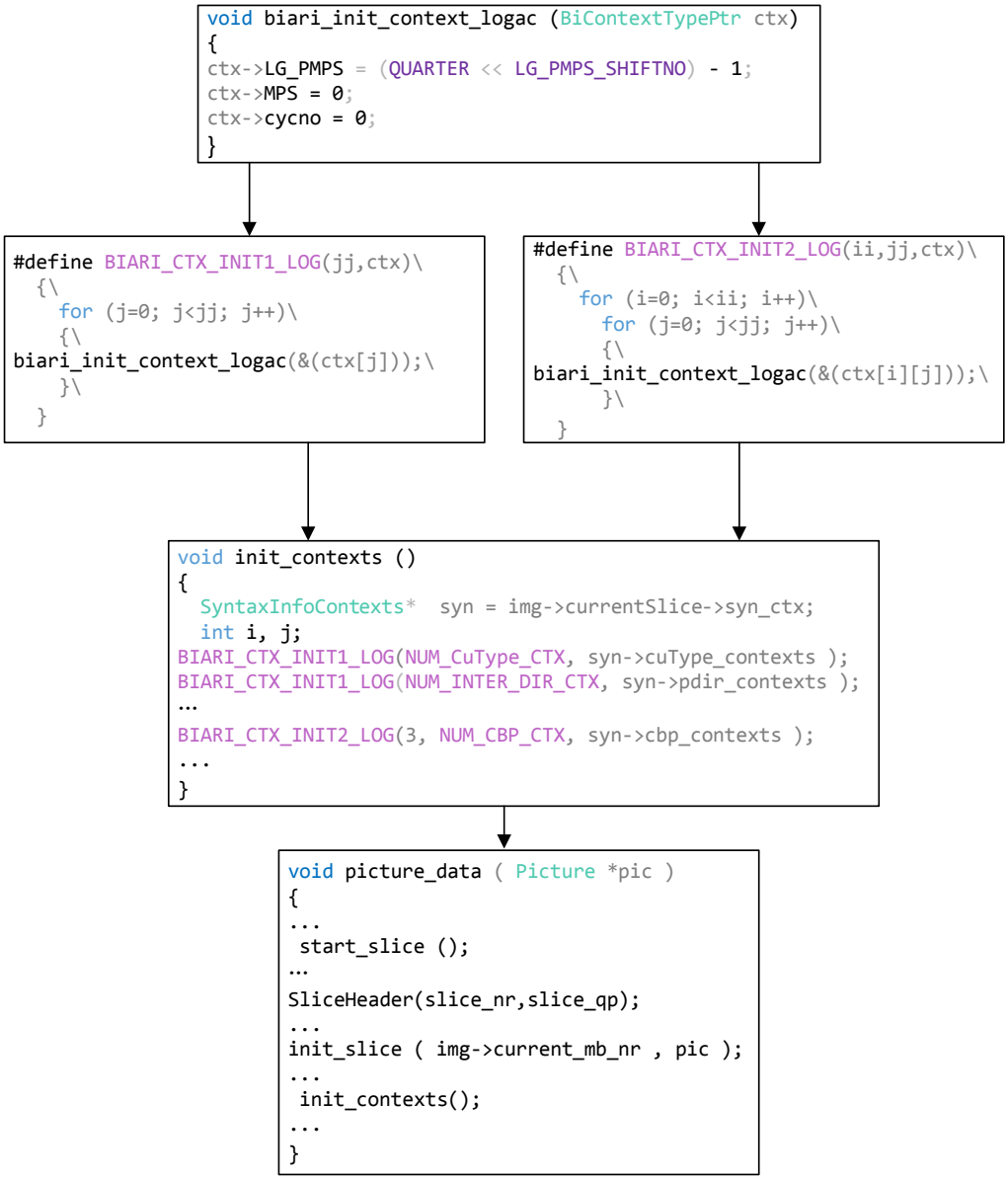

Figure 3- 2 the context initialization procedure in RD10.1

3.2.2 Adaptive Initialization Probability

In HEVC, the adaptively initial probability operation is performed by setting each context model of each syntax element an initial value and through several steps of speculative computations to get the initial probability value. However, in order to give the similar adaptation to CABAC which is implemented into our test model and then compare the coding efficiency with model using CBAC. Since the residual data accounts for the significant part (about 70%) [15] of total syntax elements, and we also know the fact that when the *LgPmps* is closer to 1023, the better compression result is, since the probability of a given symbol is about 0.5 when there is no previous symbol for current symbol to refer to. Before exploring the exactly adaptive initial probability by training numerous video sequences, the initial probability *LgPmps* for the residual data is assigned as the same as the CBAC with the same value. While for the other syntax elements, we assigned the initial value for *LgPmps* based on the following roles (3-1),

$$LgPmps'_{init} = \begin{cases} 1023 - i & i = 0,1,2,...,\lceil N_{ctx} / 2 \rceil \\ 1023 + i & i = 0,1,2,...,N_{ctx} - \lceil N_{ctx} / 2 \rceil \end{cases} \quad (3\text{-}1)$$

where N_{ctx} is the total number of context model for a given syntax element as shown in Table 3-2 and *inc* denotes the increment for the adaptive initial probability for each context. Note that when the probability value for the current context is greater than 1023, the symbol value will be given a conversion. Though this method, the initial value of probability for each syntax element will distribute near 1023 in both sides.

3.3 Experiment Result

In this section, we will analyze the performance difference in the two entropy coders. However, the performance of CABAC is measured based on above the specific initialization for some contexts. Specifically, the initial probability for each context is not identical, which is given the respective initial value for these syntax elements as described in section 3.2. And then measure the coding efficiency of CABAC modified with this initialization method. Although it is not the exact the adaptive initialization, it also give the hint that the coding efficiency trend when the context models are initialized with the distinct values.

Table 3-3 gives the coding performance result of CABAC compared with CBAC in AVS2.0. The reference is common test condition in AVS2.0 [28] and for five 1080p video sequences including *Kimono*, *ParkScene*, *Cactus*, *BasketballDrive*, and *BQTerrace* in Random Access (RA) configuration. From the result of Table 3-3, using CABAC achieves about 0.4% performance degradation compared with that of CBAC in AVS2.0. Similarly, there are also some others' work [20][21] have been proved that it is a little bit disadvantage when CABAC algorithm is used as the entropy coder in HEVC platform since the implementation detail in CABAC adopts the pre-designed look-up table where many approximations are introduced to get the pre-defined tables. While using CBAC where the logarithm domain addition/shift and domain conversion are operated can be benefit from more accurate speculations. In addition, adaptive sliding window size and adaptive probability estimation enhance the performance as

well. This work gives the conclusion that using CBAC achieves a better compression performance.

Table 3- 3 the performance comparison result of CABAC with CBAC

sequence	RDOQ off			RDOQ on		
	Y BD-rate	U BD-rate	V BD-rate	Y BD-rate	U BD-rate	V BD-rate
Kimono	0.61%	0.54%	0.54%	0.59%	0.55%	0.72%
ParkScene	0.58%	0.55%	0.42%	0.57%	0.57%	0.55%
Cactus	0.24%	0.43%	0.15%	0.32%	0.21%	0.57%
BasketballDrive	0.14%	0.35%	0.39%	0.17%	0.48%	0.10%
BQTerrace	0.29%	0.39%	0.12%	0.01%	0.36%	0.38%
Avg.	**0.37%**	**0.45%**	**0.32%**	**0.33%**	**0.43%**	**0.46%**

3.4 Conclusion

In this chapter, the proposed comparison scheme for CBAC and CABAC shows that the CBAC achieves more compelling compression performance with about 0.4% BD-rate reduction in average in RA configuration. The reason that using CBAC can achieve a better compression performance when encoding the same video sequences lies in the computation complexity of CBAC tend to be greater than that in CABAC. Domain conversion, data operation divided into integer and fraction part and comparison between integer and fraction respectively increase the calculation cost. However, more compelling coding efficiency can be obtained from these traits as the experiment result shows.

Chapter 4 CBAC Performance Improvement

Through description in the above several chapters, it has been showed that the computation complexity and sequential operation put a thread on the performance improvement. In this chapter, we will propose three ideas to improve performance of the CBAC including approximation error compensation, modification of probability estimation model and introduction of fast rate estimation to replace the real CBAC in the rate distortion optimization (RDO) process. More details for each improvement idea will be described in the following sections.

4.1 Approximation Error Compensation

As the description before, in order to simplify the computation and implementation, there are two approximation equations adopted in the domain converting process to realize the free-multiplication operation. However, the approximated error is inevitable once the approximation equations are used in the domain conversion process. Thus the error compensation method in this subclause is introduced to minimize the approximation error by domain conversion.

4.1.1 Error Compensation Table

According to the approximation principle of the Taylor's Formula, the approximation equations implemented into CBAC practically are represented as the following:

$$2^x \approx 1+x \quad (0<x<1) \tag{4-1}$$

$$\log_2(1+x) \approx x \quad (0<x<1) \tag{4-2}$$

These approximation equations are used to combine operations of both real domain and logarithmic domain, which is to replace multiplications with additions. The followings are cases of using these approximation equations:

(1) When the symbol is MPS, the range updating is performed with the *LgPmps*. While the probability update is based on the probability in the original domain, which should be derived from the *LgPmps*. Thus the approximation (4-1) is served as the bridge to draw the updating principle through *LgPmps*.

(2) When the symbol is LPS, the new range in logarithm should be derived from the original domain, where both the old and current range can be obtained from the logarithm value of each. Thus the approximation (4-1) is adopted.

(3) When the symbol is LPS, after the range updating and renormalization, there is a crucial step of the range map to prepare the logarithm-based value of the current range in order to make the parameters ready for the next iteration. Thus the approximation (4-2) is used for the transition from the original to the logarithm domain for the LPS range.

It can be see that the approximation equations defined in (4-1) and (4-2) are based on the index and logarithm of 2, though the fact is that these equations are true only when the base is *e* in the mathematical theory. Thus the approximated error induced in the process of domain conversion results in considerable performance degradation if there is no extra supplementary method to make up this. Therefore, the modification of the

approximation can be considered to minimize error in the conversion process. However, the gain which can be obtained by compensation of the approximation error will be a little bit marginal due to the incorrect probability estimation caused by the unstableness of information source. The correction function Δ1 and Δ2 can be defined as the following:

$$\Delta_1(x_1) = 1 + x_1 - 2^{x_1} \qquad (0 < x_1 < 1) \tag{4-3}$$

$$\Delta_2(x_2) = \log_2(1 + x_2) - x_2 \quad (0 < x_2 < 1) \tag{4-4}$$

Here, the x_1 and x_2 are 8 bit precision and the correction function also based on the 8 bit precision as well. The implementation is realized by indexing the pre-defined table with size of 64 where the index is 8-bit *LgPmps*. And the correction function table can be varied as the bit precision (depth) is changed. The correction factor can be quantized as the following:

$$\delta_1(index) = 2^{bitdepth}(1 + \frac{index}{64}) - 2^{bitdepth} \times 2^{\frac{index}{64}}, \qquad index = 0,1,2,...,63 \tag{4-5}$$

$$\delta_2(index) = 2^{bit\,\mathrm{depth}} \times \log_2(1 + \frac{index}{64}) - 2^{bitdepth} \times \frac{index}{64}, \qquad index = 0,1,2,...,63 \tag{4-6}$$

Here, *bitdepth* denotes the bit precision and the *index* is the needed table size. Table 4-1 shows the correction table based on the (4-5) and (4-6) and gives the difference with error adjusting table in [25]. Generally, 6 bits is enough to correct the approximation error caused by the above (4-1) and (4-2) two approximation equations.

Different from the fact that there is no exact derivation and experiment result in [25], our method gives the derivation exactly from the approximation equations and

implement into AVS2.0 in detail. In addition, only one table in [25] is adopted for both approximation equations, while our method give the exact correction table for both in Table 4-1.

Table 4- 1 The approximation error compensation table

Index	**δ1(index)*(8-bit)***	**error**	**δ2(index) *(8-bit)***	**error**	**[25]**
0	0	0	0	0	**0**
1	1.212342771	1	1.726160135	2	**2**
2	2.394329945	2	3.364894556	3	**4**
3	3.545630971	4	4.918832757	5	**5**
4	4.665911699	5	6.390487360	6	**7**
5	5.754834340	6	7.782260935	8	**8**
6	6.812057427	7	9.096452338	9	**9**
7	7.837235774	8	10.33526259	10	**10**
8	8.830020438	9	11.50080037	12	**11**
9	9.790058673	10	12.59508707	13	**12**
10	10.71699390	11	13.62006160	14	**13**
11	11.61046564	12	14.57758477	15	**14**
12	12.47010950	12	15.46944344	15	**15**
13	13.29555713	13	16.29735442	16	**16**
14	14.08643615	14	17.06296803	17	**17**
15	14.84237014	15	17.76787153	18	**17**
16	15.56297856	16	18.41359229	18	**18**
17	16.24787675	16	19.00160074	19	**19**
18	16.89667584	17	19.53331318	20	**20**
19	17.50898276	18	20.01009442	20	**20**
20	18.08440011	18	20.43326023	20	**20**
21	18.62252620	19	20.80407965	21	**21**
22	19.12295496	19	21.12377720	21	**21**
23	19.58527588	20	21.39353494	21	**21**
24	20.00907401	20	21.61449437	22	**21**
25	20.39392985	20	21.78775833	22	**22**
26	20.73941935	21	21.91439266	22	**22**
27	21.04511384	21	21.99542789	22	**22**
28	21.31057998	21	22.03186075	22	**22**
29	21.53537972	22	22.02465564	22	**21**
30	21.71907022	22	21.97474603	22	**21**
31	21.86120383	22	21.88303573	22	**21**
32	21.96132803	22	21.75040018	22	**21**
33	22.01898537	22	21.57768760	22	**20**
34	22.03371341	22	21.36572009	21	**20**

35	22.00504469	22	21.11529474	21	**21**
36	21.93250664	22	20.82718458	21	**20**
37	21.81562156	22	20.50213958	21	**20**
38	21.65390654	22	20.14088754	20	**20**
39	21.44687341	21	19.74413496	20	**19**
40	21.19402870	21	19.31256784	19	**19**
41	20.89487354	21	18.84685252	19	**19**
42	20.54890365	21	18.34763637	18	**19**
43	20.15560926	20	17.81554852	18	**18**
44	19.71447502	20	17.25120055	17	**17**
45	19.22498000	19	16.65518714	17	**17**
46	18.68659759	19	16.02808666	16	**16**
47	18.09879544	18	15.37046179	15	**15**
48	17.46103539	17	14.68286005	15	**15**
49	16.77277345	17	13.96581438	14	**14**
50	16.03345968	16	13.21984363	13	**13**
51	15.24253817	15	12.44545304	12	**12**
52	14.39944694	14	11.64313475	12	**11**
53	13.50361791	14	10.81336821	11	**11**
54	12.55447680	13	9.956620637	10	**10**
55	11.55144307	12	9.073347407	9	**9**
56	10.49392988	10	8.163992476	8	**8**
57	9.381343979	9	7.228988742	7	**7**
58	8.213085668	8	6.268758416	6	**6**
59	6.988548714	7	5.283713367	5	**5**
60	5.707120282	6	4.274255459	4	**4**
61	4.368180866	4	3.240776873	3	**3**
62	2.971104211	3	2.183660416	2	**2**
63	1.515257245	2	1.103279814	1	**0**

In addition, the approximation error compensation tables can be implemented in the encoder and similarly in the decoder part, the same compensation table is used to decode the bits generated by the modified CBAC encoder. Also, make sure that the engine should be make some definitions in the value domain limitation of engine parameters thus the encoder and decoder will be performed without overload or deadlock since this correction table can make *LgPmps* overload the minimum value.

For example, when after correcting, *LgPmps* may equal to 0, thus the deadlock will be encountered. Therefore, the specific definition should be included in code.

4.1.2 Experiment Result

Through the proposed approximation error compensation table, as experiment result in Table 4-2 shows, there will be about 0.2% in the *Luma* component and a more promising result in the *Chroma* components (about 0.3%) in average in five 1080p video sequences under Random Access (RA) configuration.

Table 4- 2 The coding efficiency using approximation error correction tables

Image seq.	Y BD-rate	U BD-rate	V BD-rate
Kimono	-0.17%	-0.54%	-0.21%
ParkScene	-0.29%	0.08%	-0.45%
Cactus	-0.26%	-0.23%	-0.47%
BasketballDrive	0.06%	-0.45%	-0.15%
BQTerrace	-0.30%	-0.48%	-0.29%
Avg.	-0.19%	-0.33%	-0.31%

4.2 Probability Estimation Model Optimization

4.2.1 Probability Estimation

The probability model updating is the crucial feature in the efficiency improvement in the arithmetic entropy coder in the video coding standard due to offering the probability of each symbol to adapt the internal state of the coder to the underlying source statistics [29]. Such adaptation enhances the compression efficiency of various entropy coding schemes such as M coder, PIPE. One of the most frequently used formulas is as the equation (4-7) shows:

$$p_{\delta}(i) = \alpha \cdot y(i) + (1-\alpha) \cdot p_{\delta}(i) \quad (4\text{-}7)$$

Here, *i* is valued as "0" or "1" which denotes that current bin is most probability symbol (MPS) or the least probability symbol (LPS), respectively. In addition, y(*i*) is 0 if the current symbol is MPS and it is assigned as 1 if otherwise. The δ denotes the probability state. The α is the scaling factor which adjusts the speed of the adaptation as it indicates that how many the in-prior encoded bins are needed to estimate the distribution of probability for the coming bins. From this recursive equation, we can get the clue that the probability updates based on α that is derived from the number of consecutive bins N_{bin} , which is defined a reciprocal number of scaling factor α. ($\alpha \leftarrow 1/N_{bin}$). The larger N_{bin} is, the speed of the adaptation is slower due to the smaller *a*, while the estimation model is more accurate. Otherwise, there will be fast transition along with a less compelling accuracy. Therefore, the choice of the referred symbols *N* determines trade-off between the model sensitivity and accuracy. About the referred bins N_{bin}, one method is always using N_{bin} bins all over the engine performing statically, while another one adopts an adaptive scaling factor *cw*, thus the referred bins can be expressed as 2^{cw}, discretely and adaptively.

Many research works focus on how to optimize the binary arithmetic coding. In [30], the proposed "virtual sliding window" method provided a more outstanding compression rate compared with look-up table index based entropy coder. Currently, the virtual sliding window technique is widely explored in HEVC. An integrated window sizes technique is introduced in [31] ~ [33], which gives a higher precision

estimation model with around 0.8% performance improvement in HEVC. In [34], a counter-based window sizes scheme is proposed and brings about 0.9% BD-rate saving. Therefore for probability estimation, the smaller window size of each probability model in the beginning of the sequence can improve the R-D performance considerably and the changeable window size tends to be more effective. The entropy coder CBAC used in AVS2 made the similar affords to design an adaptive probability estimation model to improve R-D performance, although it causes computation complexity increase.

Generally, according to how to choose window size and the probability smoothing trend, two probability distribution functions are employed including exponent mesh and uniform mesh [32][33]. The exponent mesh explains that the probability transition is based on the map function $p_\delta = 0.5(1-\alpha)^\delta$, where δ is the quantized state to realize the probability change within the certain domain, e.g. from 0.01875 to 0.5. Using this model, the practical implementation can be performed based the finite state machine (FSM) indexed by δ, thus the speculative calculation can be eliminated. However, note that the probability distribution with an exponent mesh illustrates that the probability is more dense near 0 and sparse close to 0.5. Therefore, when the probability is distributed near 0.5, there tend to be a considerable error in the evaluation. Another map function, mesh function, adopts a uniform model where the probability is scaled into certain integer section thus it can be presented as $p_\delta = P_\delta / 2^k$. The parameter k denotes the scaled range with length of 2^k and P_δ is the integer within this range. Here δ is a token for the virtual sliding window operation as there is no exactly state will be used for the

transition while the speculative computation is performed with shift or/and addition operation.

With the consideration of computation complexity and hardware-friendly in logic, the look-up table and scaling strategies are served for the practical implementation scheme of probability transition without multiplication. In table-based method, the probability updating is performed based on a pre-defined Finite State Machine (FSM), i.e. $nextState[\ ]$, where each state implies the real probability. The states jumping rules is based on the Mesh function. Benefit from the Uniform Mesh map function, another method is aimed to free multiplication with the addition and shift operation. Thus the scaled P_δ expressed as integer ranged from 1 to 2^k makes the arithmetic operations easily. The transition rules can be performed as ($P_\delta \leftarrow P_{\delta-1} \pm \{\Delta, (P_{\delta-1} \gg cw)\}$) with LPS and MPS, respectively. Here, Δ is the increment of the Uniform Mesh.

According to the required variables α , p_δ and δ in equation (4-7), the supported theories and implementation approaches employed in each video standard or relative technique to realize probability estimation are summarized in Table 4-3.

Table 4- 3 The model variables for the probability estimation

variable	models	formula	note
α	static	$1/N_{bin}$	[19]
	adaptive	$1/2^{cw}$	[25]
p_δ	exponent mesh	$p_\delta = 0.5 \times (1-\alpha)^\delta$	[33]
	uniform mesh	$p_\delta = P_\delta / 2^k$	[33]
δ	table-based	$nextState[\delta - 1]$	[19]
	scale-based	$P_{\delta-1} \pm \{\Delta, (P_{\delta-1} \gg cw)\}$	[25]

Practically, the tradeoff of the computation complexity, memory requirements and the estimation accuracy is the key problem that the implementation of probability estimation model should consider in practice. Therefore, the implementation schedule of each standard explores the method balanced all the variables and achieve the most significant performance enhancement.

4.2.2 Probability Estimation Model in CBAC

The probability estimation in AVS2.0 is performed with logarithm addition and shift operation as the CBAC algorithm employs the logarithm domain–based arithmetic coder. The Uniform Mesh and speculation computation are used for the probability update with multiplication free logic. The scaling factor for CBAC is defined as ($\alpha \leftarrow 1/2^{cw}$) with adaptive parameter *cw* chosen one of among 3, 4 and 5 according to the engine execution counter *cycno* for each context. Specifically, at the beginning several iterations, a smaller scaling factor is assigned and it will fixed at 5 after 2 iterations. In addition, the implementation of the probability estimation procedure adopts the Uniform Mesh where the scaled probability is represented as the corresponding *LgPmps* with *k*-bit resolution. Here, *LgPmps* denotes the scaled absolute value of *log2*(*Pmps*) with *Pmps* valued from (0.5, 1). Hence, the factor *k* defined in Uniform Mesh function indicts the resolution (bit-depth) of *LgPmps*, theoretically. The scaled MPS probability *LgPmps* is described as equation (4-8):

$$LgPmps = 2^{bitDepth} \times |\log_2 p_{mps}| \qquad (4\text{-}8)$$

where bit depth *bitDepth* is assigned 10-bit and *Pmps* valued from (0.5, 1). Then we can achieve two boundary values, i.e., (0, 1024), for the *LgPmps* calculation in the arithmetic coding process. Thus the probability transition can be mapped into a scaled integer range with integer operations. Specifically, the estimation updating model employed in AVS2.0 can be fulfilled in the equation (5):

$$LgPmps = \begin{cases} LgPmps - (LgPmps >> cw) & if\ \ mps \\ LgPmps + \Delta & if\ \ lps \end{cases} \tag{4-9}$$

where *cw* is the sliding window factor as described before, Δ is the increment of the *LgPmps* once encoding one bin based on the Uniform Mesh for case that the symbol is LPS case. It is also relative to the *cw* and the bit depth of the *LgPmps.*

Probability estimation is a crucial step in arithmetic coding of CBAC as illustrated in Fig.1-3. It has much influence on the final coding performance. In CBAC, context variables are included 10-bit *LgPmps*, 1-bit *valMps*, and 2-bit *cycno*. Once the arithmetic coding for a regular bin is finished, the context variables will be updated including *LgPmps* speculation, *valMps* conversion (if necessary), and cycno marking. Even through this adaptation increase the computation complexity, the coding performance of CBAC tends to be competitive compared with CABAC.

4.2.3 The Optimization of Probability Estimation Model in CBAC

In this section, based on the mechanism in CBAC, we propose an optimized probability estimation model with well-regulated scheme to improve coding efficiency.

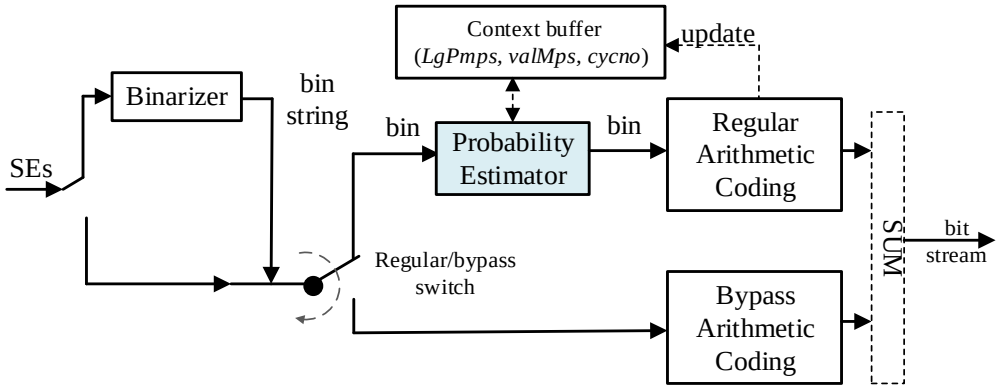

Figure 4- 1 The flowchart of CBAC encoder

Referring to the analysis of Uniform Mesh in above section, it can be concluded that the scaled probability *LgPmps* is valued within a scaled domain as $(0, 2^{bitDepth})$ in theory. Thus the probability estimation can be performed by addition or subtraction, and shift within integer data domain. Considering that the estimation error of probability of MPS near to 0.5 tends to be more considerable than that close to 1 where the difference between two symbols is marginal, we deign a feasible data domain, called (Thr_{LgPmps}, $Init_{LgPmps}$), for probability estimator of the CBAC. Thr_{LgPmps} denotes the low boundary that the scaled probability *LgPmps* can reach. $Init_{LgPmps}$ is the initial value assigned to each context model at the beginning of new slice.

For the initial value, it is assigned as in CBAC as follow equation (4-10),

$$Init_{LgPmps} = 2^{bitDepth} - \tau \qquad (4\text{-}10)$$

where τ is valued as 0 or 1. For the threshold value Thr_{LgPmps}, it is represented by (4-11):

$$Thr_{LgPmps} = 2^{bitDepth} \cdot |\log_2(1 - \hat{p}_{\min, plps})| \qquad (4\text{-}11)$$

where $\hat{p}_{\min, pls}$ is the statistical result of minimum LPS probability which can be obtained through the similar method used for the CABAC in [5]. In theory, it is a statistical result.

Based on the provided scheme, the scaled probability *LgPmps* can be transited within the feasible domain with the uniform increment each iteration in the LPS case. However, note that the adaptive scaling factor *cw* is introduced in CBAC where the sliding window size will be changed along with context variable *cycno* marking, thus the uniform increment will also adaptively change and the adaptive uniform increment $\tilde{\Delta}$ is defined as equation (4-12):

$$\tilde{\Delta}_{bitDepth,cw} = 2^{bitDepth} \times |\log_2(1-2^{-cw})| \tag{4-12}$$

Therefore, the proposed probability estimation model can be modified with the following equation (4-13):

$$LgPmps = \begin{cases} \max((LgPmps - LgPmps >> cw), Thr_{\lg pmps}) & if\ \ mps \\ LgPmps \geq 1024\,?(2^{bitDepth+1} - LgPmps) : (LgPmps + \tilde{\Delta}) & if\ \ lps \end{cases} \tag{4-13}$$

In implementation, parameters adjustments including *cw*, *bitDepth*, Thr_{LgPmps}, and $Init_{LgPmps}$ are necessary in order to find out the best scheme. Then the overall schedule for the probability estimation can be illustrated as Fig.4-2.

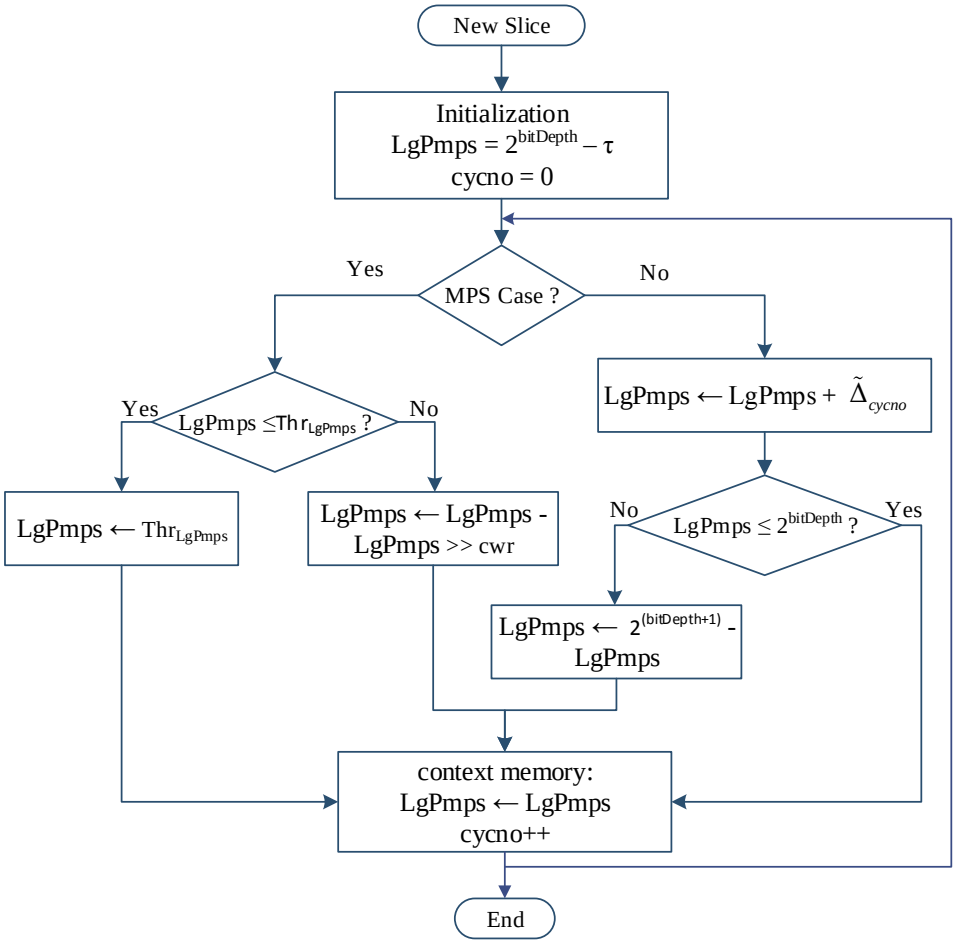

Figure 4- 2 The proposed probability estimation scheme for each context model.

4.2.4 Experiment Result

In this section, the coding efficiency enhancement result will be shown. However, for the adaptive probability estimation method, it is easy to implement with the considerable performance enhancement. To verify the coding efficiency of proposed optimized probability estimation model, experiments are conducted on RD 10.1. The bit depth *bitDepth* is assigned as 9 bits, $\hat{p}_{\min,pls}$ is about 0.0382, τ is set as 1, and the final sliding factor *cw* is set as 5. Note that *cw* is determined by the *cycno* marking and we assign the value of *cw* along with different *cycno* and syntax element type. Until *cycno* increases up to 3, *cw* is assigned 5 constantly for each context model for all syntax

elements. Table 4-4 and Table 4-5 give the BD-rate reduction detail in the A, B class video sequences under the Random Access (RA) configurations with common test condition [28] of AVS2.0.

Table 4- 4 The BD-rate of proposed probability estimation with RDOQ-off

Size	**Sequence**	**Y BD-rate**	**U BD-rate**	**V BD-rate**	**Avg.**
A Class	*Traffic*	-0.20%	-0.24%	-0.71%	-0.27%
	Pku-girls	-0.10%	-0.94%	-0.73%	-0.28%
	PeopleOnStreet	-0.21%	-0.88%	-1.43%	-0.45%
B Class	*ParkScene*	-0.05%	-0.43%	-0.63%	-0.17%
	beach	-0.07%	-8.48%	-9.65%	-2.32%
	taishan	-0.13%	-0.25%	-0.62%	-0.21%
	kimono	-0.10%	-0.21%	-0.47%	-0.16%
	cactus	-0.28%	-0.43%	-0.70%	-0.35%
	BasketballDrive	-0.29%	-0.67%	-0.54%	-0.37%
Avg.		**-0.16%**	**-1.35%**	**-1.76%**	**-0.52%**

Table 4- 5 The BD-rate of proposed probability estimation with RDOQ on.

Size	**Sequence**	**Y BD-rate**	**U BD-rate**	**V BD-rate**	**Avg.**
A Class	*Traffic*	-0.23%	-0.39%	-0.76%	-0.32%
	Pku-girls	-0.16%	-0.44%	-0.42%	-0.22%
	PeopleOnStreet	-0.23%	-0.73%	-0.89%	-0.37%
B Class	*ParkScene*	-0.09%	-0.50%	-0.15%	-0.15%
	beach	-0.14%	-7.37%	-6.00%	-1.78%
	taishan	-0.15%	-0.53%	-0.45%	-0.23%
	kimono	-0.19%	-0.00%	-0.32%	-0.19%
	cactus	-0.24%	-0.53%	-0.18%	-0.27%
	BasketballDrive	-0.13%	-0.47%	-0.62%	-0.24%
Avg.		**-0.17%**	**-1.22%**	**-1.09%**	**-0.42%**

4.3 Rate Estimation

4.3.1 Rate Estimation Model

In AVS2.0, it is crucial to find out the efficient rate distortion optimization (RDO) mode decision for enhancing the coding efficiency. This mode decision is aimed to selects an optimal mode among various available candidates including supported size of coding unit, the prediction unit and the transform unit. However, the rate distortion cost in RD10.1, reference s/w of AVS2.0 is obtained from the rates coming from the real CBAC instead of using rate estimation table which is used in HM, reference s/w of [35]. In addition, based on the previous several research[36][37][38], we proposed the fast rate estimation model for AVS2.0 to replace the real CBAC since the process of CBAC tend to be complicated because of the serial nature and strong data dependence. In this section, we will describe the proposed rate estimation (RE) model for the rate estimation in the RDO mode decision process implemented with RE table and Fig.4-3 illustrates the rate estimation idea in the AVS2.0 where we use the pre-defined RE table to replace the real CBAC engine to calculate the rate distortion cost.

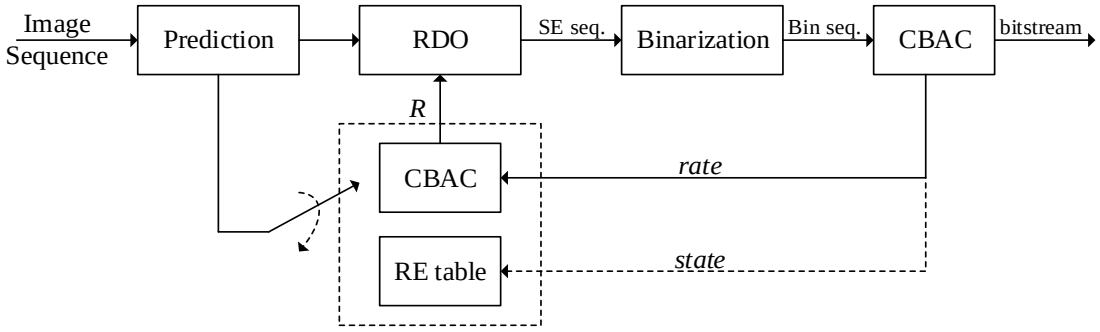

Figure 4- 3 The block diagram of proposed rate estimation

Current rate estimation is built based on the fact that there is a relationship between the

probability and range as Fig. 4-4 shows. The statistics tell us that the probability of each new range can be described as (4-14):

$$p(r) = \frac{k_0}{r} \tag{4-14}$$

Here, r denotes for the new obtained range and k_0 is the constant. Thus, according the range r varies from 256 to 510 in theory, the constant k_0 can be derived and it is presented with $log_2 e$. Therefore, the rate estimation model dedicated with estimated bits can be further built.

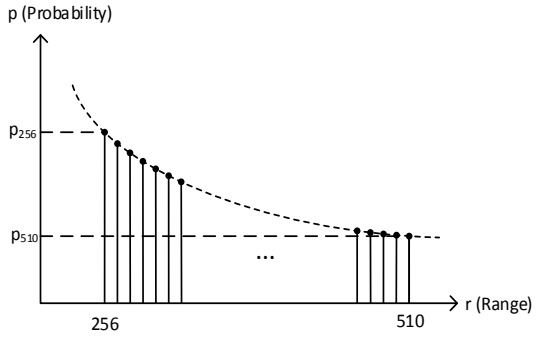

Figure 4- 4 Probability distribution of the CABAC range

Based on this probability distribution function, the expected output bit length is represented with (4-15) if the input bin is the least probability symbol (LPS). Otherwise, (4-16) is adopted.

$$l_{LPS}(s) = \sum_{x=256}^{510} \log_2 \frac{x}{R(s)} \cdot \frac{\log_2 e}{x} \tag{4-15}$$

$$l_{MPS}(s) = \sum_{x=256}^{510} \log_2 \frac{x}{x - R(s)} \cdot \frac{\log_2 e}{x} \tag{4-16}$$

Here $R(s)$ denotes the value of range indexed by context state s. Therefore, the expected bit length for both MPS and LPS case defined in the above equations can be basic model

for the distinguished arithmetic coding engine.

In addition, the rate model for CBAC which uses logarithm adder and shift will be deduced as described in the following. In this model, *LgPmps* denotes the MPS probability in logarithm domain with 10-bit precision. Therefore, the corresponding probability P_{mps} in original can be derived from (4-17).

$$\begin{aligned} \log p_{mps} &= -LgPmps / 2^{10} \\ p_{mps} &= 2^{-LgPmps/1024} \end{aligned} \tag{4-17}$$

In the principle of arithmetic coding on logarithm, all the related parameters are derived from probability of MPS in logarithm domain, which is *LgPmps*. Therefore, the expected bit length of a bin can be achieved as (4-18) if input bin is MPS, on the contrary, (4-19) is derived.

$$\begin{aligned} l_{mps}(LgPmps) &= \sum_{x=256}^{510} \log_2 \frac{x}{R_i} \cdot \frac{k_0}{x} = \sum_{x=256}^{510} \log_2 \frac{x}{x \cdot p_{mps}} \cdot \frac{k_0}{x} \\ &= -\log_2 p_{mps} \cdot \sum_{x=256}^{510} \frac{k_0}{x} \\ &= \frac{LgPmps}{1024} \end{aligned} \tag{4-18}$$

$$\begin{aligned} l_{lps}(LgPmps) &= \sum_{x=256}^{510} \log_2 \frac{x}{R_i} \cdot \frac{k_0}{x} = \sum_{x=256}^{510} \log_2 \frac{x}{x \cdot p_{lps}} \cdot \frac{k_0}{x} \\ &= -\log_2 p_{lps} \cdot \sum_{x=256}^{510} \frac{k_0}{x} \\ &= -\log_2 (1 - 2^{-\frac{LgPmps}{1024}}) \end{aligned} \tag{4-19}$$

From (4-18) and (4-19), the estimated bit length is achieved indexing by the *LgPmps*. However, the bit length tend to be changed with the bit depth of *LgPmps*. We also

designed experiments that verify the effects of bit depth of *LgPmps* (fraction part of bit length) in the rate estimation RE table.

4.3.2 Experiment Result

To verify the coding efficiency of RD10.1 encoder with proposed rate estimation model, we use the AVS2.0 common test condition [28] for five 1080p video sequences including *Kimono*, *ParkScene*, *Cactus*, *BasketballDrive*, and *BQTerrace* in Random Access (RA), All Intra (AI), and Low Delay P (LDP) configurations.

Table 4-7 shows the coding performance after using rate estimation table with 2-bit fraction part and 8-bit fraction part. Note that the same rate estimation table with 8-bit fraction part is used for rate distortion optimization quantization (RDOQ). Fig.4-5 illustrates the coding performance in BD-rate (%) varying according to the change of fraction part from 2-bit to 8-bit. We can get the conclusion that the coding efficiency tend to be almost constant when the fraction part is larger than 2-bit. There are about 0.1% BD-rate reduction in RA, a marginal (0.02%) increase under AI and a slight performance degradation with 0.18% in LDP configuration. This trend keeps the similar between 2-bit and 8-bit in AVS2.0. Thus it is important to know that the rate estimation table should be at least 2-bit fraction part to implement the correct rate estimation model in the RDO process. In addition, Table 4-8 gives the encoding time saving when the rate estimation is implemented into AVS2.0 for the RD cost calculation in the RDO process. There is about 1.24% encoding time reduction compared with the original AVS2.0 reference software.

Table 4- 6 The BD-rate of using rate estimation (2-bit and 8-bit fraction part)

1080p image seqence	RE table (8-bit)			RE table (2-bit)		
	RA	AI	LDP	RA	AI	LDP
Kimono	0.03%	0.00%	0.10%	0.10%	0.03%	0.13%
ParkScene	0.03%	-0.04%	0.01%	0.12%	-0.02%	0.14%
Cactus	-0.06%	-0.02%	0.37%	-0.08%	-0.04%	0.09%
BasketballDrive	-0.14%	0.13%	0.23%	-0.15%	0.09%	0.05%
BQTerrace	-0.38%	0.02%	0.22%	-0.27%	0.03%	0.46%
Average	-0.10%	0.02%	0.18%	-0.06%	0.02%	0.17%

Table 4- 7 The time saving when the rate estimation table is used in AVS2.0

Test seq.	QP	Anchor time	Rate est. Time	Time Saving	Avg.
Kimono	27	4989.17	4967.22	-0.44%	**-1.02%**
	32	4730.05	4653.46	-1.62%	
	38	4607.79	4562.96	-0.97%	
	45	4100.52	4056.9	-1.06%	
ParkScene	27	3594.39	3560.48	-0.94%	**-1.16%**
	32	3233.44	3198.90	-1.07%	
	38	2982.04	2948.08	-1.14%	
	45	2817.84	2776.41	-1.47%	
Cactus	27	3203.02	3155.65	-1.48%	**-1.68%**
	32	2959.27	2909.86	-1.67%	
	38	2897.11	2876.14	-0.72%	
	45	2579.71	2506.22	-2.85%	
BasketballDrive	27	10375.7	10265.3	-1.06%	**-1.10%**
	32	7762.71	7822.48	-0.77%	
	38	8873.24	8957.54	-0.95%	
	45	7960.32	8090.07	-1.63%	
Average					**-1.24%**

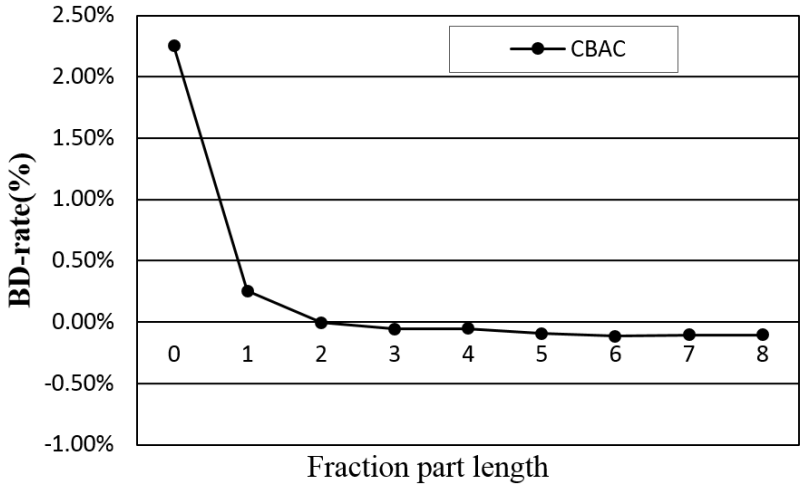

Figure 4- 5 The BD-rate changes with different fraction part lengths

4.4 Conclusion

In this section, ideas for improving performance in terms of the engine optimization and throughput improvement were described in detail. From the experimental results, we can obtain three conclusions. One is approximation error modification is aimed to match the arithmetic coding principle without the approximation operation. There is 0.2% BD-rate improvement in the *Luma* component at the sake of addition of a 2-D buffer to store the adjusting factor and increase of a little of computation.

Another work is about the probability estimation. Since the performance analysis shows that the bit resolution of *LgPmps* tend to affects the coding efficiency, the proposed probability estimation model using 9-bit resolution with matched parameters achieve a better performance with about 0.3% BD-rate saving in average.

Lastly, for rate estimation, we can see that there is at least 2 bits fraction part for rate estimation RE table when implementing the rate estimation in RDO process.

Chapter 5 Implementation of Binary Arithmetic Decoder in CBAC

Because of serial data dependency of the process of updating range and context probabilities in a CBAC algorithm, it is still challengeable to implement decoder of CBAC with high throughput.

There are numerous previous work [39] [40] which have been devoted to improve the throughput for CABAC encoder in HEVC. [39] shows various methods to improve the throughput including grouping bypass bin, reducing the context data dependence, and sharing context modeling, etc. Several hardware-orientated tools such pre-renormalization, hybrid path coverage, bypass bin splitting, were developed for the binary arithmetic encoding for HEVC in [40]. In addition, the recent publication [41] researched on the architecture of CBAC encoder in AVS1 targeting to the real-time HDTV applications. However, plenty of works are CABAC encoder/decoder in HEVC, although there are several literatures about AVS, most of them are for CAVLD of AVS1. Few work is about CBAC architecture design, especially CBAC decoder.

Generally, the overall CBAC decoder includes several steps: binary arithmetic decoder (BAD), context updating and selection and debinarization. In this chapter, we design Binary Arithmetic Decoder, with throughput of one-bin per cycle which is a part of CBAC Decoder and a main bottleneck of accomplishing high throughput by strong data dependency. Specifically, BAD includes range update, offset update and bit read when one bin is decoded. The most important work of this chapter is designing a reasonable

critical path of BAD.

5.1 Architecture of BAD

The difficulty of implementation of CBAC decoder with high throughput lies in the high serial data dependencies from several update loops: range update, offset update, and context update. For the introduction of the operation of logarithm domain to free multiplication, a variable of range is represented as 2 terms, which are *RangeI* for integer part and *RangeF* for fractional part of range. The representation of Offset is the same as that of range. We design the conceptive structure including the main loops needed in the decoder part. And the general CBAC decoder implementation structure can be described as Fig.5-1. There are several loops which are performed with strong data dependency in CBAC Decoder and each loops are marked with different colors.

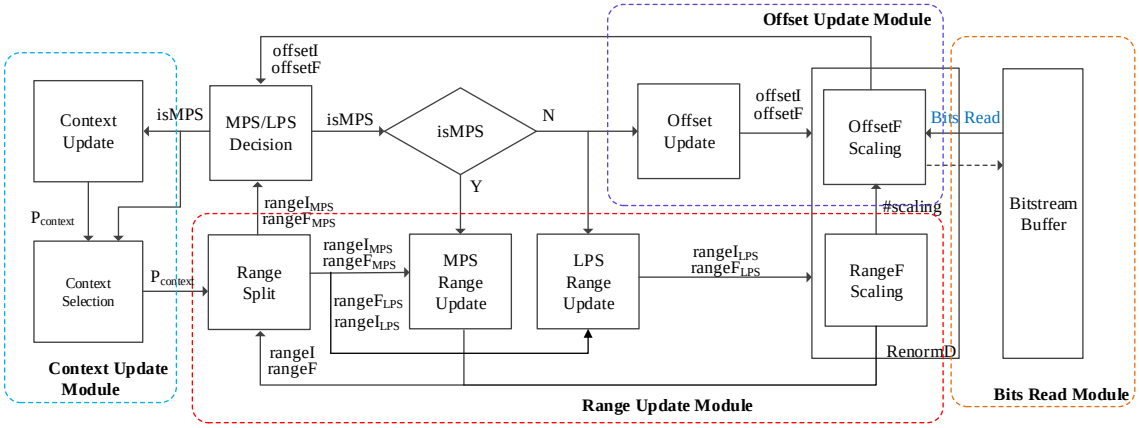

Figure 5- 1 the General BAD Structure in AVS2.0

This overall structure can be divided into four sub-structures including range update module, offset update module, bits read module and context update module when implemented.

5.1.1 Top Architecture of BAD

For BAD structure design, there are three loops we should consider: range update, offset update and bits read. One-bin per cycle scheme requires all the loops to be performed in the one clock. Then signals for the interface between each module are need to be matched when one bin is performed. Each module will be given details of the design through the block diagram and Verilog code logic.

In this top architecture, range update module is firstly performed based on the algorithm design in CBAC, followed by the offset update module using the output signals of range update module including the fraction part of MPS symbol *o_rangeFMps*, integer part of LPS symbol *o_rangeILps*, and the LPS symbol *isLps*. The bits read module reads bits from bit-stream and uses the signals generated by offset update module indicts how many bits that the bits read module should obtain from the bit-stream buffer. Though this procedure, one bin is decoded and the parameters including range and offset are updated and prepared for the next bin. The overall structure can be described as Fig.5-2 where the interface signals are given the detailed illustration.

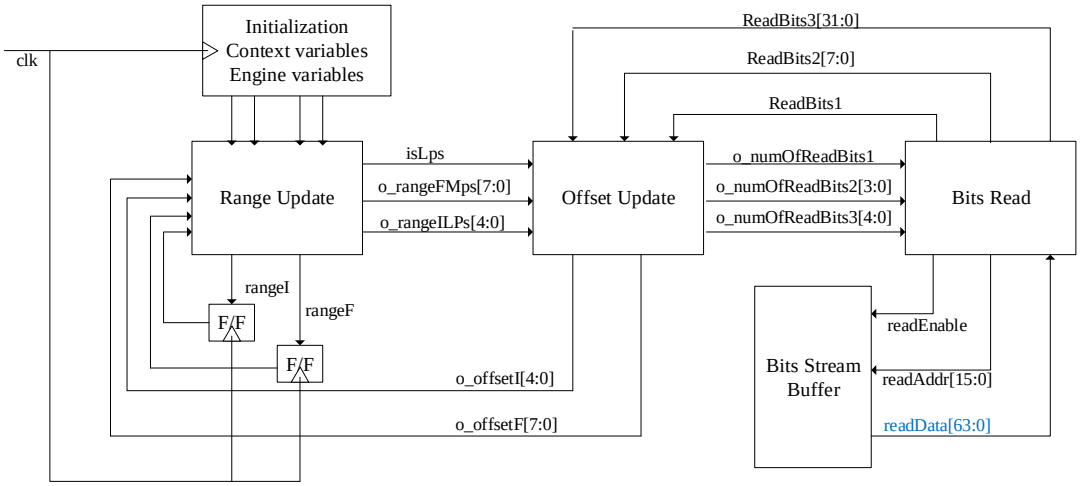

Figure 5- 2 The overall structure for the BAD with one-bin scheme

5.1.2 Range Update Module

Firstly, the range update procedure is described in this section. Range Update for MPS and LPS case is the similar as that in encoder part with integer and fraction part. In order to make the update scheme clear, the integer part of range *rangeI* and fraction part *rangeF* are divided into respective sub-module. The flow charts of updating *rangeI* and *rangeF* are shown as in Fig. 5-3 and Fig. 5-4, respectively. For the integer part, if it is LPS symbol, *rangeI* will be changed as the *rangeF* which decides the increment of *rangeI* as *rangeF* should be scaled until it is not smaller than 256. However, after finishing all the scaling, the *rangeI* is assigned as 0 again in LPS case.

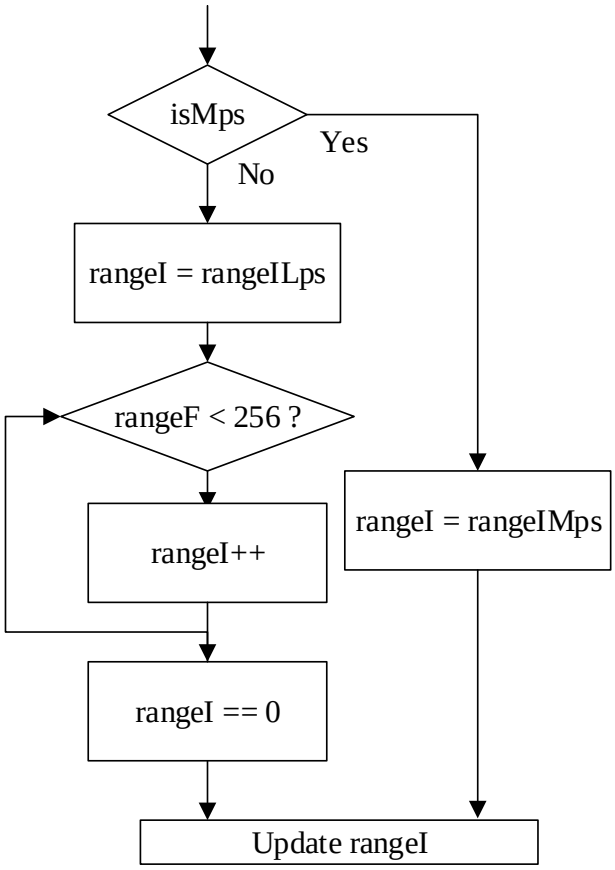

Figure 5- 3 Flow chart of *rangeI* update

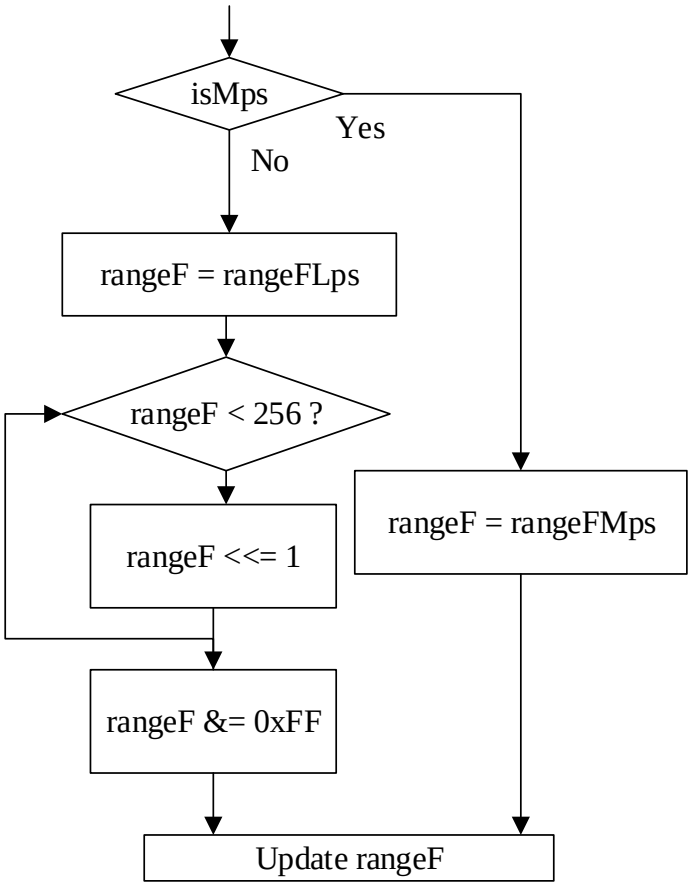

Figure 5- 4 Flow chart of *rangeF* update

For the fraction part of range *rangeF* is performed with the similar stage as above *offsetI* update after re-modifying the original code in RD10.1. Fig.5-4 shows the process of updating *rangeF* where the similar scaling and shift operations are performed but the difference is that the *rangeF* is scaled using the 8-bit of low significant bit (LSB) in LPS case. Based on the *rangeF* and *rangeI* update scheme in C code, then the range update procedure can be described as the following Fig.5-5 where two LPS scaling modules are included into the range update module, which describes two cases whether the integer parts are equal or not. Once the process of updating *rangeI* and *rangeF* are finished, updated value of each variable is stored to register (F/F), which is for next process.

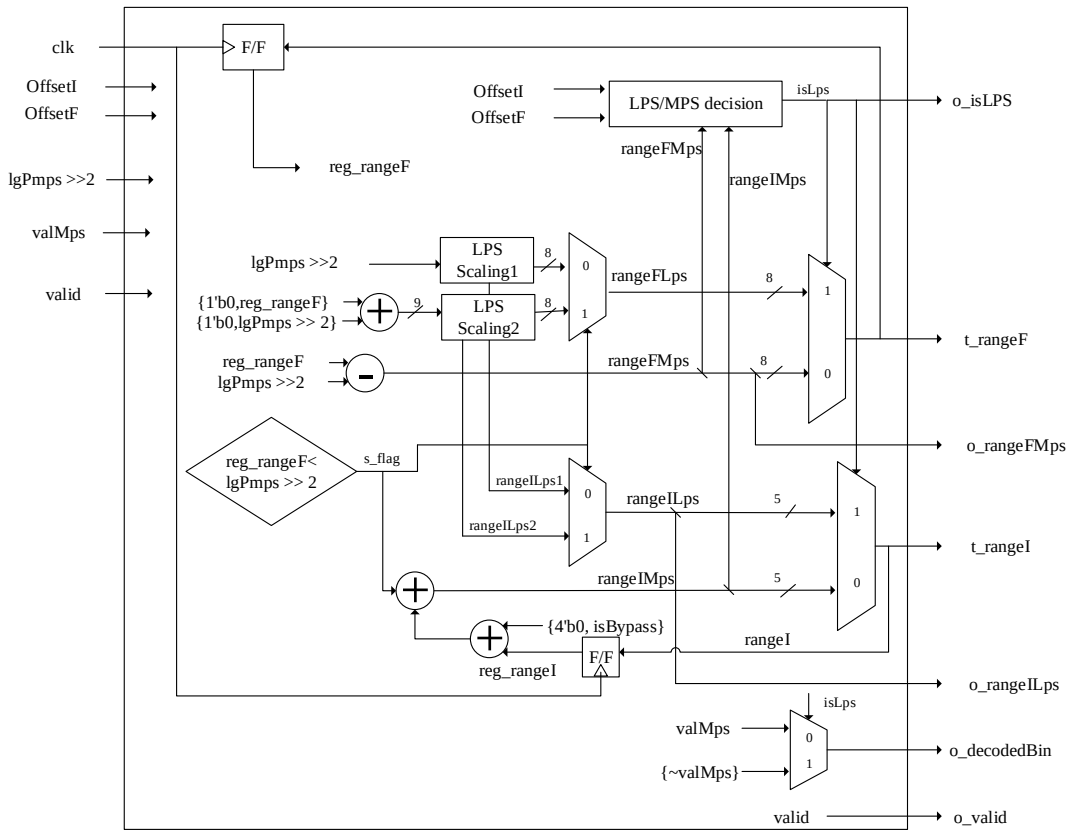

Figure 5- 5 Detailed Structure of Module for Range Update

5.1.3 Offset Update Module

In this section, the offset update module is introduced with the design detail. Range Update plays the vital role in the sub-range division process. According to the speculation of range and offset in the decoder part, the offset update is performed with the intermediate result of the process of updating range such as *rangeILps* and *rangeFMps.* By comparing rangeFMps with *offset*, we can decide whether decoded bin is MPS or not. After the MPS/LPS decision is made, *offset* update is performed. Since there is no offset update in MPS case, thus the offset update is performed in the LPS case only. The flow chart in Fig.5-6 illustrates of the process of updating *offsetI* where *offsetI* keeps the same without updating when it is MPS case.

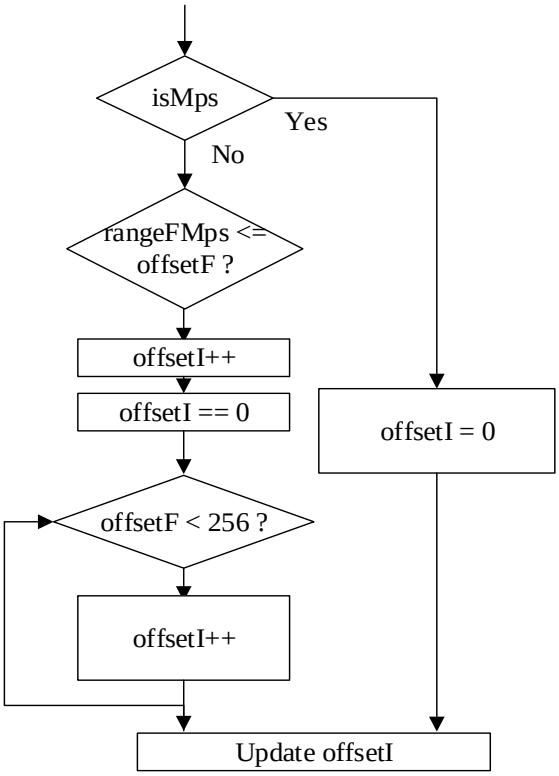

Figure 5- 6 offsetI update block diagram

Then we will analysis the offsetF update in the following section. In the offset update module, it is updated the value of offset in case of LPS only. There are *offsetF* scaling and *rangeF* scaling for the bit reads and range updating gives the hint how many bits should be read to decide the *offsetF*. The flow chart for the offsetF update can be described as Fig.5-7.

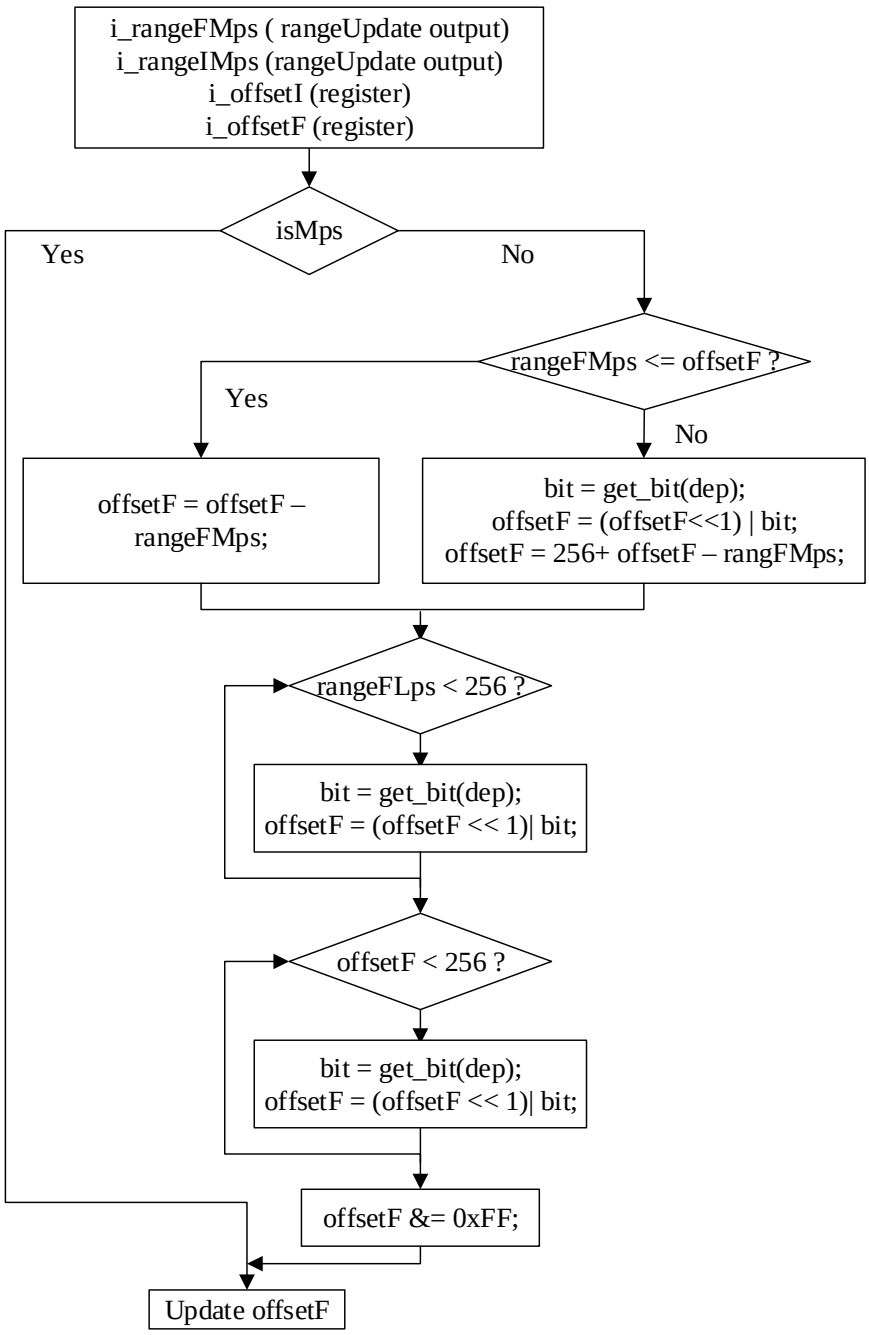

Figure 5- 7 flow chart of updating *offsetF*

Finally, through the algorithm analysis, the offset update module can be described as Fig.5-8. The input signals includes *rangeILps*, *rangeFMps* and *isLps*, which are generated after finishing range update module, and reads some other signals relative to bit read module that will be described in bit read module.

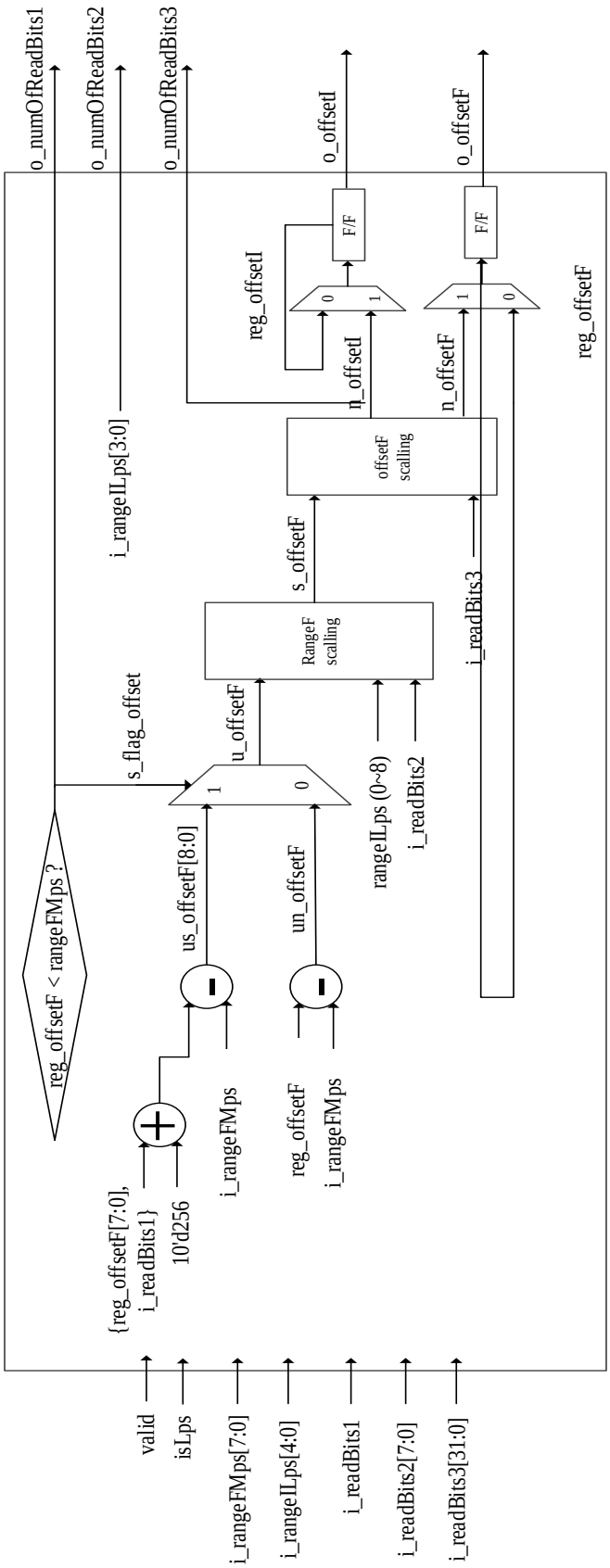

Figure 5- 8 Offset Update logic diagram block

5.1.4 Bits Read Module

The bit read module defines the bits read operation in BAD. As shown in Fig 5-7, the bit read and offset update is performed with the jointly procedure. Part of the input signals of offset update come from the output variables including o_readBits1, o_readBits2 and o_readBits3, which indicts different bit channel.

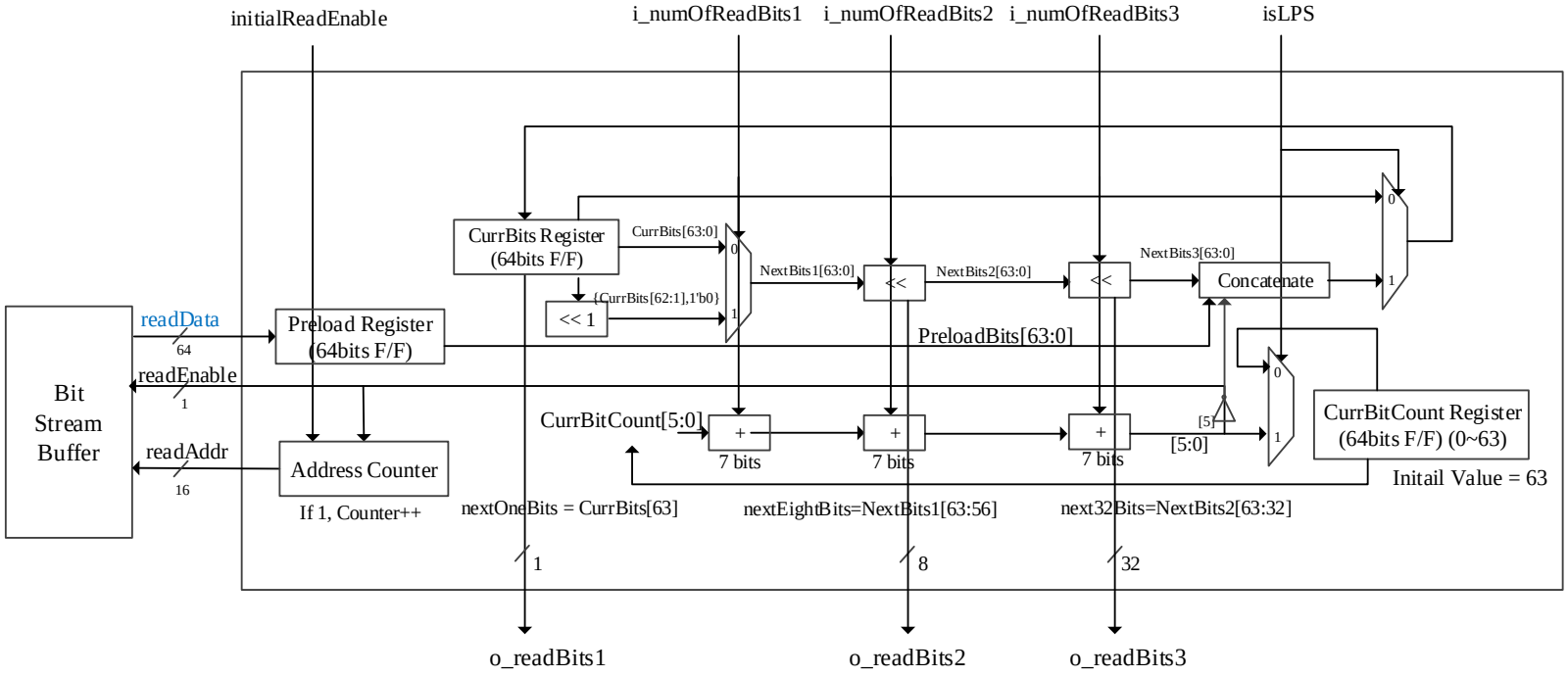

Figure 5- 9 Bits Read Logic Block Diagram

5.1.5 Context Modeling

In this section, the context update is described although context update module is not included Binary Arithmetic Coding in CBAC. It is because it is one of bottle-neck of implementing design with high throughput by context data dependency, which means consecutive bins with same context index should be decoded in a sequence. The variables for context update in CBAC *are LgPmps, which is* a variable for context probability, *cycno, which is* a variable for sliding window parameter, and *valMps*, *which is* a flag indicating whether current decoded bin is MPS or not. In the CBAC decoder, all the context variables are assigned with the same value – *LgPmps* is 1023, *cycno* equals to 0 and *valMps* initialized as 0 at the beginning of the new slice. Specifically, the process of updating context variables is designed as the following Fig.5-12. Once the context updating is finished, the context model for the current bin is updated with the new variable for the next access within current slice.

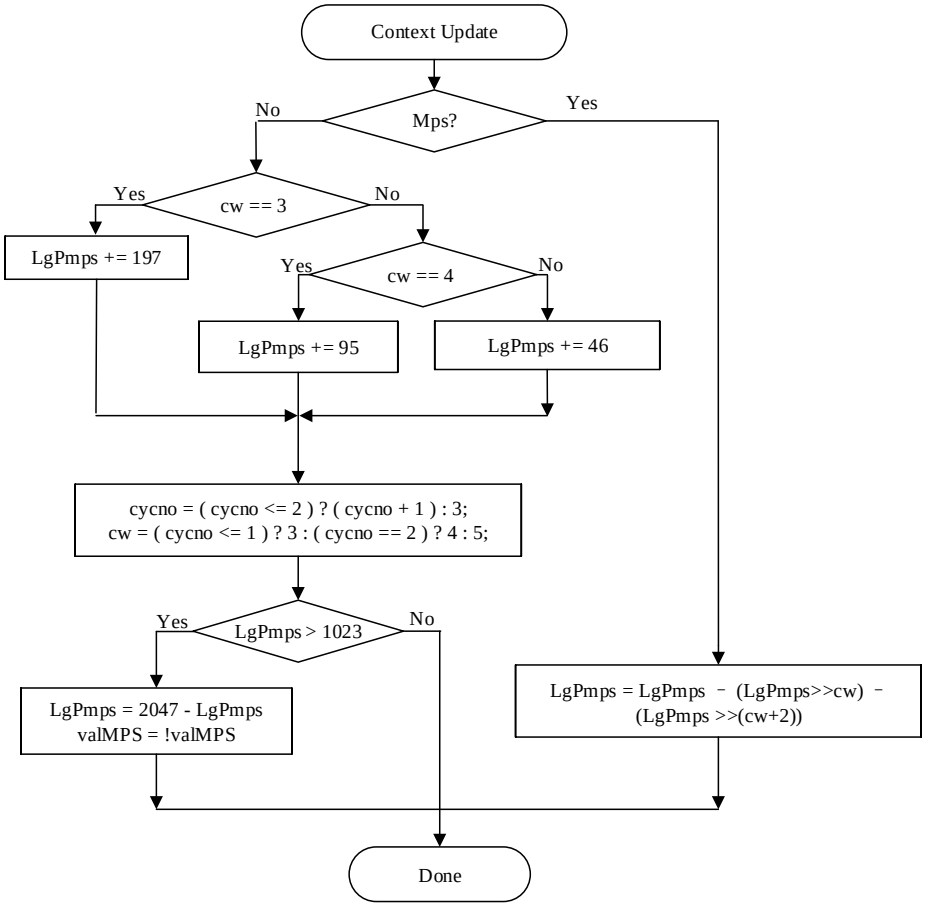

Figure 5- 10 The process of Context Updating in the CBAC decoder in AVS2.0

For the hardware design, signals for interface should be defined clearly. For the *cycno* update, and *cycno* is related to the sliding window factor *cw*, which is relative the sliding window size in the probability update process. The *valMps* is changed only when *LgPmps* is larger than 1024, which means probability is out of the defined bit precision (10 bits) and *valMps* should be reversed (0 –> 1 or 1 –> 0). According to this analysis, the block diagram for context update can be described with the following architecture in Fig. 5-11.

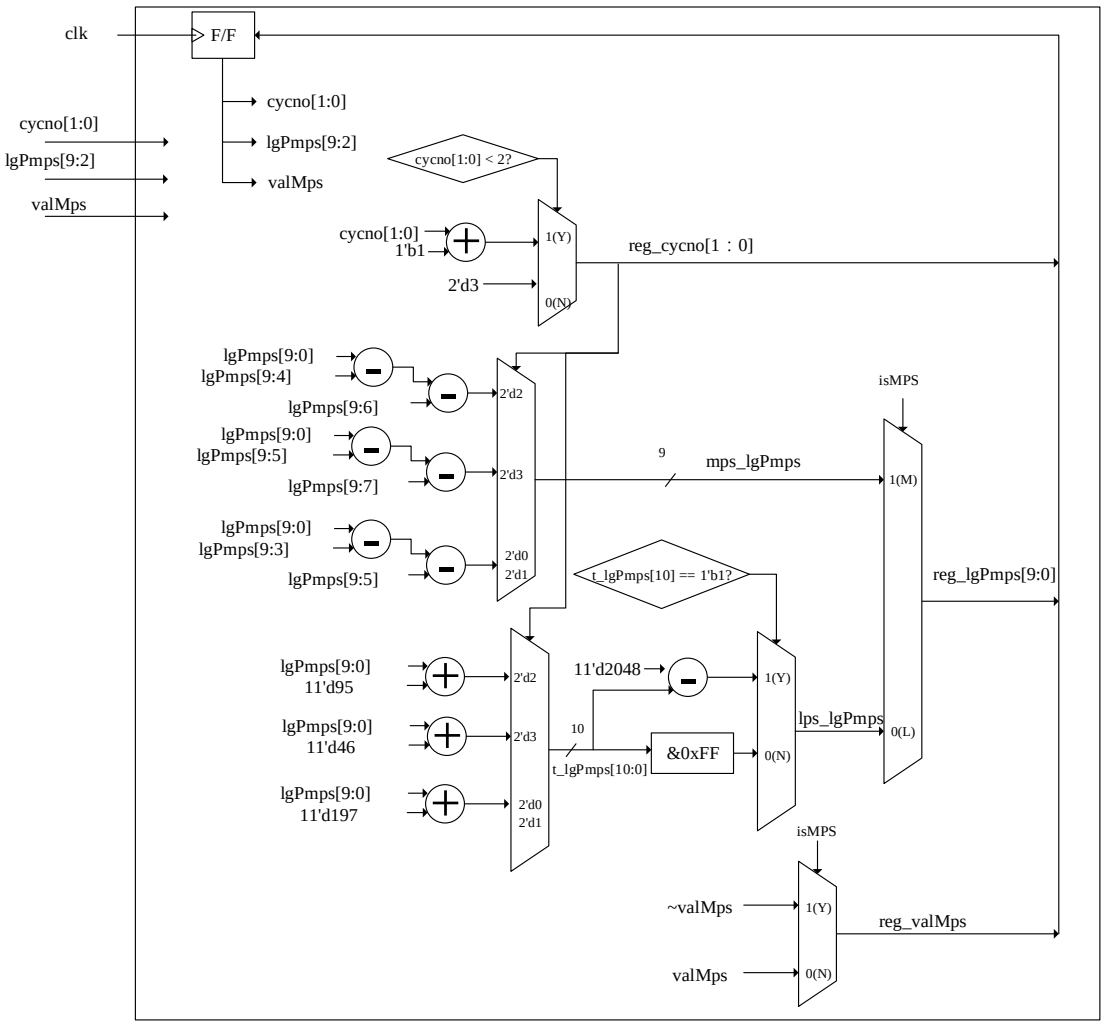

Figure 5- 11 Detailed Structure of Module for Context Update

5.2 Complexity of BAD

This design is synthesized using the TSMC 65 nm LP process. From the synthesis result, we can see that the critical path of this design is related to paths to update *offsetF* in case of LPS.

At synthesis level, it achieves a maximum clock rate of 526 MHz. So we can expect that this design has an operating frequency of more than 400 MHz in the level of chip in consideration of overhead by place and routing with the margin of 20%, which is a kind of estimated figure by experiential knowledge and depends on competence level

of engineer, which deal with CAD tools for place and routing. And the total gate count is about 13.3K, which is including BAD only.

There is no issued research results about the one-bin per cycle design for AVS2.0, most of researches are based on the first generation AVS1.0, HEVC or H.264/AVC. Although [44] has been designed for CBAC in AVS1, it is also available to compare with proposed BAD design. Since the different synthesis processors are used, after normalizing the frequencies [45][46] collected by our design and [44], the comparison detail can be shown in Table 5-1.

Table 5- 1 Summary of the implementation result

	[3]	Ours
Standard	AVS1.0	AVS2.0
Process technology	0.18um CMOS	TSMC 65nm LP
Max. frequency (Synthesized)	150 MHz	526 MHz
Total gate count	21.5 k	-
BAD only (excluding bitstream Control)	6.3k	6.7k
Throughput	1 bin/cycle	1 bin/cycle

5.3 Conclusion

There is no significant changes in CBAC decoder algorithm from AVS1.0 to AVS2.0. We propose an architecture for Binary Arithmetic Decoder in CBAC, which is crucial part of implementing whole of CBAC Decoder with high throughput. Although we focus on implementing BAD with throughput of one bin per cycle, it is possible to extend this design to the architecture for multi-bin decoding in considering the fact that there is no offset update in MPS case. It means we can improve throughput of this

design if we can decode multiple MPS Bins at a time without increasing delay of critical paths.

In the current stage, this one-bin scheme obtains the basic BAD engine and it will be a premising exploration for the multi-bin design in order to improve the throughput for the real-time applications or surveillance camera. In addition, implementation of the context update and debinarization are not achieved in this stage of this research topic for lack of time. In the near future, the context update and de-binarization will be given much consideration based on the BAD design in this thesis. In addition, based on this design, we can explore the multi-bin scheme in future as well.

Chapter 6 Conclusion and Further Work

6.1 Conclusion

In this dissertation, the author performed three aspects works on the entropy coding CBAC of AVS2.0 including CBAC performance analysis, Arithmetic Coder engine optimizations and the CBAC decoder architecture implementation.

In the performance analysis chapter, we concluded that CBAC achieves a better performance under the proposed comparison scheme even though CABAC transplanted in RD10.1 with the adaptive initial context models at the beginning of each new slice. Since the adaptive probability estimation and adaptive sliding window size adjusting methods are introduced into CBAC to map the source information for the given video sequence, the performance is proved that CBAC has the better compression performance compared with CABAC. The CBAC optimization is another topic in this thesis work.

Based on the each parameters used in CBAC, the relative exploration is performed, especially in the approximation error optimization and probability estimation re-scalability. Though verifying the best bit depth of the scaled probability *LgPmps*, the various bit resolutions are tested and then get the conclusion that 9-bit resolution with the relative parameters setting can achieve a significant efficiency enhancement. Actually, CBAC adopts various variables both in engine parameters and context variables, only these variables are trained very well via numerous adjusting, the CBAC

can achieve the considerable algorithm simplification and performance improvement. Otherwise, it is difficult to get more progress.

For the CBAC decoder implementation, the author explores the hardware performance though proposed one-bin per cycle architecture. Firstly, modify the C code in RD code into hardware design language Verilog code and design the one-bin scheme including range update, offset update, bits read, and context update and debinarization logics. Then match and verify the Verilog code and C code though comparing the simulation result. Finally, analyze the hardware architecture performance. For this one-bin scheme design, the maximum frequency is up to 526 MHz in theory and the total gate count is about 13.3K based on the technique TSMC 65 process.

6.2 Future Works

For the future works, there are two aspects which are challengeable to achieve more progress in the coding efficiency. Firstly, simplifying the CBAC encoder/decoder logic, especially in the update loops with the serial data domain conversion. It can be referred as the algorithm optimization based on the software RD code of AVS2.0 since CBAC logic still accounts for the considerable computation complexity. Thus exploring a more simplified scheme without much performance degradation is one of the further effort needed to spare to. Another is the implementation for the multi-bin schedule which is aimed to improve the throughput, especially for the ultra-high definition video or the real-time applications. As the growing requirements on the video information, such as TV programs, on-line movie, surveillance camera, etc., in daily life, the high throughput

architecture tend to be more compelling and only the efficient multiple bins architecture can make it come true. Therefore, multi-bin architecture for CBAC decoder will be proposed and designed in the future.

Reference

[1] B CCITT S, Recommendation H. 261-Video Codec for Audio visual Services at px64 Kbit/s. The International Telegraph and Telephone Consultative Committee, 1990.

[2] ITU-T. H.263. Video Codec for Low Bit Rate Communication. 1996.

[3] Le Gall D. MPEG: A Video Compression Standard for Multimedia Applications. Communications of the ACM, 1991, 34(4): 46-58.

[4] Draft I. recommendation and final draft international standard of joint video specification (ITU-T Rec. H. 264| ISO/IEC 14496-10 AVC).

[5] Information Technology – High Efficiency Media Coding – Part2: Video, Chinese GB/T200602, 2006.

[6] B. Bross, W.-J. Han, J.-R. Ohm, G. J. Sullivan, and T. Wiegand, "High efficiency video coding (HEVC) text specification draft 10," Joint Collaborative Team on Video Coding (JCT-VC) of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11, document JCTVC-L1003, Geneva, Switzerland, Jan. 2013.

[7] G. J. Sullivan, J.-R. Ohm, W. J. Han and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard", IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, pp. 1649-1668, Dec. 2012.

[8] Information Technology – High Efficiency Media Coding – Part2: Video, Chinese GB/T201503, 2015.

[9] Gao W, Ma S. An Overview of AVS2 Standard, Advanced Video Coding Systems. Springer International Publishing, 2014: 35-49.

[10]余全合, 曹潇然, 李蔚然, 荣耀程, 何芸, 郑萧桢, 郑建铧, “短距离帧内预测技术”, AVS_M3171, 沈阳，2013 年 9 月.

[11]凌勇，朱兴国，虞露， J. Chen，S. Lee, Y. Piao, C. Kim，“一种前向双假设预测模式”，AVS_M3271，深圳，2013 年 12 月.

[12]I. Kim, S. Lee, Y. Piao and C. Kim, "Directional multi-hypothesis prediction (DMH) for AVS2”, 45th AVS meeting, AVS_M3094, Taicang, Jun. 2013.

[13]马俊铖，马思伟，安基程，张凯，雷少民，“渐进的运动矢量精度”，AVS_M3049，洛阳，2013 年 3 月.

[14]Y. Piao, S. Lee, A. Saxena, C. Kim, “Secondary transform for intra coding”, 47th AVS meeting, AVS_M3233, Shenzhen, Dec. 2013.

[15]J. Wang, X. Wang, T. Ji and D. He, "Two-level transform coefficient coding," 43rd AVS meeting, AVS_M3035, Beijing, Dec.2012.

[16]Jie Chen, Sunil Lee, Elena Alshina, Chanyul Kim, Chih-Ming Fu, Yu-Wen Huang, Shawmin Lei, “Sample Adaptive Offset for AVS2”, AVS_M3197, 45th AVS meeting, Shenyang, Sep. 2013.

[17]张新峰，司俊俊，王苫社，马思伟，蔡家扬，陈庆晔, 黄毓文,雷少民，“AVS2 自适应环路滤波器”，AVS_M3292，北京，2014 年 4 月.

[18]L. Zhang, et al. "Context-based entropy coding in AVS video coding standard." Signal Processing: Image Communication 24.4 (2009): 263-276.A. Rosenfeld and A. Kak. Digital Image Processing (2nd Edition, Vol. 2 ed.),Academic Press, Orlando (1982)

[19]M. Detlev, H. Schwarz, and T. Wiegand. "Context-based adaptive binary

arithmetic coding in the H. 264/AVC video compression standard." Circuits and Systems for Video Technology, IEEE Transactions on 13.7 (2003): 620-636.

[20] E. Alshina, E. Alshin, (2011) Multi-parameter probability up-date for CABAC, Joint Collaborative Team on Video Coding (JCT-VC), Document JCTVC-F254, Torino, July 2011

[21] J. Stegemann, H. Kirchhoffer, D. Marpe, T. Wiegand, (2011) Non-CE1: counterbased probability model update with adapted arithmetic coding engine, Joint Collaborative Team on Video Coding (JCT-VC), Document JCTVC-G547, Geneva, Nov. 2011

[22] Hankerson D C, Harris G A, Johnson Jr P D. “Introduction to information theory and data compression.” CRC press, 2003.

[23] Said A. “Introduction to arithmetic coding-theory and practice”. Hewlett Packard Laboratories Report, 2004.

[24] Gao W, Ma S W, “Advanced Video Coding Systems”. Springer, 2014.

[25] Yu W, Yang P, He Y. Arithmetic Coding on Logarithm Domain.

[26] Sole J, Joshi R, Nguyen N, et al. Transform coefficient coding in HEVC. Circuits and Systems for Video Technology, IEEE Transactions on, 2012, 22(12): 1765-1777.

[27] H. Jung, S. Choi, and S–I. Chae. "Coding efficiency of the context-based arithmetic coding engine of AVS 2.0 in the HEVC encoder." Consumer Electronics (ICCE), 2015 IEEE International Conference on. IEEE, 2015.

[28] AVS-P2 common test condition, AVS-N2020, 2014.

[29] Hankerson D C, Harris G A, Johnson Jr P D. “Introduction to information theory and data compression.” CRC press, 2003.

[30] Belyaev E, Gilmutdinov M, Turlikov A (2006) Binary arithmetic coding system with adaptive probability estimation by “virtual sliding window” in IEEE tenth international symposium on consumer electronics (ISCE ’06), pp 1–5, 2006.

[31] Alshin A, Alshina E, Park J H. “High precision probability estimation for CABAC” in Visual Communications and Image Processing (VCIP), 2013. IEEE, 2013: 1-6.

[32] Alshin A, Alshina E, Park. I "CEI (subset B): Multi-parameter probability up-date for CABAC," Document of Joint Collaborative Team on Video Coding, JCTVC-G0764, November 2011.

[33] Alshina E, Alshin A (2011) Multi-parameter probability up-date for CABAC, Joint Collaborative Team on Video Coding (JCT-VC), Document JCTVC-F254, Torino, July 2011

[34] Stegemann J, Kirchhoffer H,Marpe D,Wiegand T (2011) Non-CE1: counter-based probability model update with adapted arithmetic coding engine, Joint Collaborative Team on Video Coding (JCT-VC), Document JCTVC-G547, Geneva, Nov. 2011

[35] Bossen, F.: ‘CE1: table-based bit estimation for CABAC’. JCTVC-G763, Geneva, November 2011

[36] Hahm, J., and Kyung, C.-M.: ‘Efficient CABAC rate estimation for H.264/AVC mode decision’, IEEE Trans. Circuits Syst. Video Technol., 2010, 20, (2), pp. 310–316

[37] Won, K., Yang, J., and Jeon, B.: ‘Fast CABAC rate estimation for H.264/AVC mode decision’, Electron. Lett., 2012, 48, (19), pp. 1201–1203

[38] Choi S, Chae S I. Comparison of CABAC rate estimation models for HEVC rate

distortion optimization. Electronics Letters, 2014, 50(6): 441-442.

[39] V. Sze and M. Budagavi, “High throughput CABAC entropy coding in HEVC,” IEEE Trans. Circuits Syst. Video Technol., vol. 22, no. 12, pp. 1778–1791, Dec. 2012.

[40] Zhou J, Zhou D, Fei W, et al. “A high-performance CABAC encoder architecture for HEVC and H. 264/AVC”, Image Processing (ICIP), 2013 20th IEEE International Conference on. IEEE, 2013: 1568-1572.

[41] Li Y, Zhang S, Jia H, et al. “A high-throughput low-latency arithmetic encoder design for HDTV”, Circuits and Systems (ISCAS), 2013 IEEE International Symposium on. IEEE, 2013: 998-1001.

[42] Chen Y H, Sze V. A Deeply Pipelined CABAC Decoder for HEVC Supporting Level 6.2 High-tier Applications. 2014.

[43] Yi Y, Park I C. High-speed h. 264/AVC CABAC decoding. Circuits and Systems for Video Technology, IEEE Transactions on, 2007, 17(4): 490-494.

[44] Zheng J, Gao W, Wu D, et al. An efficient VLSI architecture for CBAC of AVS HDTV decoder. Signal Processing: Image Communication, 2009, 24(4): 324-332.

[45] Dennard R H, Rideout V L, Bassous E, et al. Design of ion-implanted MOSFET's with very small physical dimensions. Solid-State Circuits, IEEE Journal of, 1974, 9(5): 256-268.

[46] Bohr M. A 30 year retrospective on Dennard's MOSFET scaling paper. Solid-State Circuits Society Newsletter, IEEE, 2007, 12(1): 11-13.

Appendix

A.1. Co-simulation Environment

In this section, the Verilog codes for each module will be shown in detail.

A.1.1 Range Update Module (dRangeUpdate.v)

```
`timescale 1ns/100ps

module dRangeUpdate (
    input                       clk,
    input                       rst_n,

    // Signals from Context Modeling
    input                       i_reset,
    input                       i_valid,
    input                       i_valMPS,
    input       [7:0]           i_lgPmps,

    // Signals from Offset Update
    input       [7:0]           i_offsetF,
    input       [4:0]           i_offsetI,

    // Signals to Context Modeling
    output                      o_valid,
    output                      o_decodedBin,

    // Signals for Updating Offset
    output                      o_isLPS,
    output      [7:0]           o_rangeFMps,
    output      [4:0]           o_rangeILps,

    // Signals for Test ,
    output      [7:0]           t_rangeF,
    output      [4:0]           t_rangeI
);
////////////////////////////////////////////////////////////////////////////////
// range Update Stage
////////////////////////////////////////////////////////////////////////////////
```

```
    wire                            isBypass        ;
    wire                            valMPS          ;
    wire    [7:0]                   lgPmps          ;

    reg                                 isLPS           ;

    wire                            s_flag          ;

    reg         [7:0]                   reg_rangeF      ;
    reg         [4:0]                   reg_rangeI      ;
    wire    [7:0]                   updated_rangeF  ;
    wire    [4:0]                   updated_rangeI  ;

    wire        [4:0]                   rangeIMps       ;
    wire        [4:0]                   rangeILps       ;
    wire        [4:0]                   rangeILps1      ;
    wire        [4:0]                   rangeILps2      ;

    wire    [7:0]                   rangeFMps       ;
    wire    [7:0]                   rangeFLps       ;
    wire    [7:0]                   rangeFLps1      ;
    wire    [7:0]                   rangeFLps2      ;

    ////////////////////////////////////////////////////////////////////////////
    // Input
    ////////////////////////////////////////////////////////////////////////////
    assign isBypass             = (i_lgPmps == 0 ) ? 1'b1 : 1'b0 ;
    assign valMPS               = i_valMPS ;
    assign lgPmps               = i_lgPmps ;

    ////////////////////////////////////////////////////////////////////////////
    // Output
    ////////////////////////////////////////////////////////////////////////////
    assign o_valid              = i_valid ;
    assign o_decodedBin         = ( isLPS == 1'b1 ) ? ~valMPS    : valMPS ;

    assign o_isLPS              = isLPS         ;
    assign o_rangeFMps          = ( isLPS == 1'b1 ) ? rangeFMps : 8'b0 ;
    assign o_rangeILps          = ( isBypass == 1'b0 && isLPS == 1'b1 ) ? rangeILps :
5'b0 ;

    assign t_rangeF             = updated_rangeF ;
    assign t_rangeI             = updated_rangeI ;
```

```
    ////////////////////////////////////////////////////////////////////////////
    // MPS/LPS Decision
    ////////////////////////////////////////////////////////////////////////////
    always@(i_offsetI,i_offsetF,rangeIMps,rangeFMps) begin
        if ( rangeIMps > i_offsetI    || (i_offsetI == rangeIMps && i_offsetF >= rangeFMps ) ) begin
            isLPS                   = 1'b1 ;
        end else begin
            isLPS                   = 1'b0 ;
        end
    end

    ////////////////////////////////////////////////////////////////////////////
    // s_flag
    ////////////////////////////////////////////////////////////////////////////
    assign s_flag               = ( reg_rangeF < lgPmps ) ? 1'b1 : 1'b0 ;

    ////////////////////////////////////////////////////////////////////////////
    // Range MPS
    ////////////////////////////////////////////////////////////////////////////
    assign rangeFMps            = reg_rangeF - lgPmps ;
    assign rangeIMps            = reg_rangeI + {4'b0,s_flag} + {4'b0,isBypass} ;

    ////////////////////////////////////////////////////////////////////////////
    // Range LPS
    ////////////////////////////////////////////////////////////////////////////
    assign rangeFLps            = ( s_flag == 1'b1 ) ? rangeFLps2 : rangeFLps1 ;
    assign rangeILps            = ( s_flag == 1'b1 ) ? rangeILps2 : rangeILps1 ;

    dLPSScaling1 A_LPSScaling1(
        .i_rangeF               ( reg_rangeF    ),
        .i_lgPmps               ( lgPmps        ),
        .rangeFLps1             ( rangeFLps1        ),
        .rangeILps1             ( rangeILps1    )
    );

    dLPSScaling2 A_LPSScaling2(
        .i_rangeF               ( reg_rangeF    ),
        .i_lgPmps               ( lgPmps        ),
        .rangeFLps2             ( rangeFLps2        ),
        .rangeILps2             ( rangeILps2    )
    );

    ////////////////////////////////////////////////////////////////////////////
```

```
    // rangeF Update
    //////////////////////////////////////////////////////////////////////////////
    assign updated_rangeF       = ( isBypass == 1'b0 && isLPS    == 1'b1 ) ? rangeFLps :
rangeFMps ;

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
            reg_rangeF              <= 8'hFF ;
        end else begin
            if ( i_reset == 1'b1 ) begin
                reg_rangeF          <= 8'hFF ;
            end else if ( i_valid == 1'b1 ) begin
                reg_rangeF          <= updated_rangeF ;
            end
        end
    end

    //////////////////////////////////////////////////////////////////////////////
    // rangeI Update
    //////////////////////////////////////////////////////////////////////////////
    assign updated_rangeI       = ( isLPS    == 1'b1 ) ? 5'b0 : rangeIMps ;

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
            reg_rangeI              <= 5'b0 ;
        end else begin
            if ( i_reset == 1'b1 ) begin
                reg_rangeI          <= 5'b0 ;
            end else if ( i_valid == 1'b1 ) begin
                reg_rangeI          <= updated_rangeI ;
            end
        end
    end

endmodule
```

In the range update module, there are two scaling operations are introduced in order to describe the operations in each case in LPS.

```
`timescale 1ns/100ps

module dLPSScaling1 (
    input       [7:0]               i_rangeF,
    input       [7:0]               i_lgPmps,
```

```
    output    reg  [7:0]                    rangeFLps1,
    output    reg  [4:0]                    rangeILps1
);

    always@(i_lgPmps,i_rangeF,i_rangeF) begin
        case(i_lgPmps)
        8'b00000000 :    rangeFLps1   = i_rangeF ;
        8'b00000001 :    rangeFLps1   = 8'b0 ;
        8'b00000010 :    rangeFLps1   = {i_lgPmps[0],7'b0} ;
        8'b00000011 :    rangeFLps1   = {i_lgPmps[0],7'b0} ;
        8'b00000100 :    rangeFLps1   = {i_lgPmps[1:0],6'b0} ;
        8'b00000101 :    rangeFLps1   = {i_lgPmps[1:0],6'b0} ;
        8'b00000110 :    rangeFLps1   = {i_lgPmps[1:0],6'b0} ;
        8'b00000111 :    rangeFLps1   = {i_lgPmps[1:0],6'b0} ;
        8'b00001000 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001001 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001010 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001011 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001100 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001101 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001110 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00001111 :    rangeFLps1   = {i_lgPmps[2:0],5'b0} ;
        8'b00010000 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010001 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010010 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010011 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010100 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010101 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010110 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00010111 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011000 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011001 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011010 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011011 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011100 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011101 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011110 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00011111 :    rangeFLps1   = {i_lgPmps[3:0],4'b0} ;
        8'b00100000 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
        8'b00100001 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
        8'b00100010 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
        8'b00100011 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
        8'b00100100 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
        8'b00100101 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
```

```
8'b00100110 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00100111 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101000 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101001 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101010 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101011 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101100 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101101 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101110 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00101111 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110000 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110001 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110010 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110011 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110100 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110101 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110110 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00110111 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111000 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111001 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111010 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111011 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111100 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111101 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111110 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b00111111 :    rangeFLps1   = {i_lgPmps[4:0],3'b0} ;
8'b01000000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01000111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01001111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
```

```
8'b01010010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01010111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01011111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01100111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01101111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01110111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111000 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111001 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111010 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111011 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111100 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
8'b01111101 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
```

```
        8'b01111110 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
        8'b01111111 :    rangeFLps1   = {i_lgPmps[5:0],2'b0} ;
        default   :    rangeFLps1   = {i_lgPmps[6:0],1'b0} ;
        endcase
    end

    always@(i_lgPmps) begin
        case(i_lgPmps[7:1])
        7'b0000000  :    rangeILps1   =   5'd8 ;
        7'b0000001  :    rangeILps1   =   5'd7 ;
        7'b0000010  :    rangeILps1   =   5'd6 ;
        7'b0000011  :    rangeILps1   =   5'd6 ;
        7'b0000100  :    rangeILps1   =   5'd5 ;
        7'b0000101  :    rangeILps1   =   5'd5 ;
        7'b0000110  :    rangeILps1   =   5'd5 ;
        7'b0000111  :    rangeILps1   =   5'd5 ;
        7'b0001000  :    rangeILps1   =   5'd4 ;
        7'b0001001  :    rangeILps1   =   5'd4 ;
        7'b0001010  :    rangeILps1   =   5'd4 ;
        7'b0001011  :    rangeILps1   =   5'd4 ;
        7'b0001100  :    rangeILps1   =   5'd4 ;
        7'b0001101  :    rangeILps1   =   5'd4 ;
        7'b0001110  :    rangeILps1   =   5'd4 ;
        7'b0001111  :    rangeILps1   =   5'd4 ;
        7'b0010000  :    rangeILps1   =   5'd3 ;
        7'b0010001  :    rangeILps1   =   5'd3 ;
        7'b0010010  :    rangeILps1   =   5'd3 ;
        7'b0010011  :    rangeILps1   =   5'd3 ;
        7'b0010100  :    rangeILps1   =   5'd3 ;
        7'b0010101  :    rangeILps1   =   5'd3 ;
        7'b0010110  :    rangeILps1   =   5'd3 ;
        7'b0010111  :    rangeILps1   =   5'd3 ;
        7'b0011000  :    rangeILps1   =   5'd3 ;
        7'b0011001  :    rangeILps1   =   5'd3 ;
        7'b0011010  :    rangeILps1   =   5'd3 ;
        7'b0011011  :    rangeILps1   =   5'd3 ;
        7'b0011100  :    rangeILps1   =   5'd3 ;
        7'b0011101  :    rangeILps1   =   5'd3 ;
        7'b0011110  :    rangeILps1   =   5'd3 ;
        7'b0011111  :    rangeILps1   =   5'd3 ;
        7'b0100000  :    rangeILps1   =   5'd2 ;
        7'b0100001  :    rangeILps1   =   5'd2 ;
        7'b0100010  :    rangeILps1   =   5'd2 ;
        7'b0100011  :    rangeILps1   =   5'd2 ;
```

```
        7'b0100100  :   rangeILps1  =   5'd2 ;
        7'b0100101  :   rangeILps1  =   5'd2 ;
        7'b0100110  :   rangeILps1  =   5'd2 ;
        7'b0100111  :   rangeILps1  =   5'd2 ;
        7'b0101000  :   rangeILps1  =   5'd2 ;
        7'b0101001  :   rangeILps1  =   5'd2 ;
        7'b0101010  :   rangeILps1  =   5'd2 ;
        7'b0101011  :   rangeILps1  =   5'd2 ;
        7'b0101100  :   rangeILps1  =   5'd2 ;
        7'b0101101  :   rangeILps1  =   5'd2 ;
        7'b0101110  :   rangeILps1  =   5'd2 ;
        7'b0101111  :   rangeILps1  =   5'd2 ;
        7'b0110000  :   rangeILps1  =   5'd2 ;
        7'b0110001  :   rangeILps1  =   5'd2 ;
        7'b0110010  :   rangeILps1  =   5'd2 ;
        7'b0110011  :   rangeILps1  =   5'd2 ;
        7'b0110100  :   rangeILps1  =   5'd2 ;
        7'b0110101  :   rangeILps1  =   5'd2 ;
        7'b0110110  :   rangeILps1  =   5'd2 ;
        7'b0110111  :   rangeILps1  =   5'd2 ;
        7'b0111000  :   rangeILps1  =   5'd2 ;
        7'b0111001  :   rangeILps1  =   5'd2 ;
        7'b0111010  :   rangeILps1  =   5'd2 ;
        7'b0111011  :   rangeILps1  =   5'd2 ;
        7'b0111100  :   rangeILps1  =   5'd2 ;
        7'b0111101  :   rangeILps1  =   5'd2 ;
        7'b0111110  :   rangeILps1  =   5'd2 ;
        7'b0111111  :   rangeILps1  =   5'd2 ;
        default  :   rangeILps1  =   5'd1 ;
        endcase
    end

endmodule
```

```
`timescale 1ns/100ps

module dLPSScaling2 (
    input       [7:0]       i_rangeF,
    input       [7:0]       i_lgPmps,
    output  reg [7:0]       rangeFLps2,
    output  reg [4:0]       rangeILps2
);
```

```
wire [8:0]                  temp ;
wire       [7:0]                sel ;

assign temp                     = {1'b0,i_rangeF} + {1'b0,i_lgPmps} ;
assign sel                      = temp[8:1] ;

always@(sel,temp) begin
    case(sel)
    8'b00000000 :    rangeFLps2   = 8'b0 ;
    8'b00000001 :    rangeFLps2   = {temp[0],7'b0} ;
    8'b00000010 :    rangeFLps2   = {temp[1:0],6'b0} ;
    8'b00000011 :    rangeFLps2   = {temp[1:0],6'b0} ;
    8'b00000100 :    rangeFLps2   = {temp[2:0],5'b0} ;
    8'b00000101 :    rangeFLps2   = {temp[2:0],5'b0} ;
    8'b00000110 :    rangeFLps2   = {temp[2:0],5'b0} ;
    8'b00000111 :    rangeFLps2   = {temp[2:0],5'b0} ;
    8'b00001000 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001001 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001010 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001011 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001100 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001101 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001110 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00001111 :    rangeFLps2   = {temp[3:0],4'b0} ;
    8'b00010000 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010001 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010010 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010011 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010100 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010101 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010110 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00010111 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011000 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011001 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011010 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011011 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011100 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011101 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011110 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00011111 :    rangeFLps2   = {temp[4:0],3'b0} ;
    8'b00100000 :    rangeFLps2   = {temp[5:0],2'b0} ;
    8'b00100001 :    rangeFLps2   = {temp[5:0],2'b0} ;
    8'b00100010 :    rangeFLps2   = {temp[5:0],2'b0} ;
    8'b00100011 :    rangeFLps2   = {temp[5:0],2'b0} ;
```

```
8'b00100100 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00100101 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00100110 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00100111 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101000 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101001 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101010 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101011 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101100 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101101 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101110 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00101111 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110000 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110001 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110010 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110011 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110100 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110101 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110110 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00110111 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111000 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111001 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111010 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111011 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111100 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111101 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111110 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b00111111 :    rangeFLps2   = {temp[5:0],2'b0} ;
8'b01000000 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000001 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000010 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000011 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000100 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000101 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000110 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01000111 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001000 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001001 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001010 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001011 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001100 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001101 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001110 :    rangeFLps2   = {temp[6:0],1'b0} ;
8'b01001111 :    rangeFLps2   = {temp[6:0],1'b0} ;
```

```
8'b01010000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010011 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010100 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010101 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010110 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01010111 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011011 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011100 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011101 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011110 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01011111 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100011 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100100 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100101 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100110 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01100111 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101011 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101100 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101101 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101110 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01101111 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110011 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110100 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110101 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110110 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01110111 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01111000 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01111001 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01111010 :    rangeFLps2  = {temp[6:0],1'b0} ;
8'b01111011 :    rangeFLps2  = {temp[6:0],1'b0} ;
```

```
        8'b01111100 :     rangeFLps2   = {temp[6:0],1'b0} ;
        8'b01111101 :     rangeFLps2   = {temp[6:0],1'b0} ;
        8'b01111110 :     rangeFLps2   = {temp[6:0],1'b0} ;
        8'b01111111 :     rangeFLps2   = {temp[6:0],1'b0} ;
        default   :     rangeFLps2   = temp[7:0] ;
        endcase
    end

    always@(sel) begin
        case(sel)
        8'b00000000 :     rangeILps2   =    5'd8 ;
        8'b00000001 :     rangeILps2   =    5'd7 ;
        8'b00000010 :     rangeILps2   =    5'd6 ;
        8'b00000011 :     rangeILps2   =    5'd6 ;
        8'b00000100 :     rangeILps2   =    5'd5 ;
        8'b00000101 :     rangeILps2   =    5'd5 ;
        8'b00000110 :     rangeILps2   =    5'd5 ;
        8'b00000111 :     rangeILps2   =    5'd5 ;
        8'b00001000 :     rangeILps2   =    5'd4 ;
        8'b00001001 :     rangeILps2   =    5'd4 ;
        8'b00001010 :     rangeILps2   =    5'd4 ;
        8'b00001011 :     rangeILps2   =    5'd4 ;
        8'b00001100 :     rangeILps2   =    5'd4 ;
        8'b00001101 :     rangeILps2   =    5'd4 ;
        8'b00001110 :     rangeILps2   =    5'd4 ;
        8'b00001111 :     rangeILps2   =    5'd4 ;
        8'b00010000 :     rangeILps2   =    5'd3 ;
        8'b00010001 :     rangeILps2   =    5'd3 ;
        8'b00010010 :     rangeILps2   =    5'd3 ;
        8'b00010011 :     rangeILps2   =    5'd3 ;
        8'b00010100 :     rangeILps2   =    5'd3 ;
        8'b00010101 :     rangeILps2   =    5'd3 ;
        8'b00010110 :     rangeILps2   =    5'd3 ;
        8'b00010111 :     rangeILps2   =    5'd3 ;
        8'b00011000 :     rangeILps2   =    5'd3 ;
        8'b00011001 :     rangeILps2   =    5'd3 ;
        8'b00011010 :     rangeILps2   =    5'd3 ;
        8'b00011011 :     rangeILps2   =    5'd3 ;
        8'b00011100 :     rangeILps2   =    5'd3 ;
        8'b00011101 :     rangeILps2   =    5'd3 ;
        8'b00011110 :     rangeILps2   =    5'd3 ;
        8'b00011111 :     rangeILps2   =    5'd3 ;
        8'b00100000 :     rangeILps2   =    5'd2 ;
        8'b00100001 :     rangeILps2   =    5'd2 ;
```

```
8'b00100010 :    rangeILps2   =   5'd2 ;
8'b00100011 :    rangeILps2   =   5'd2 ;
8'b00100100 :    rangeILps2   =   5'd2 ;
8'b00100101 :    rangeILps2   =   5'd2 ;
8'b00100110 :    rangeILps2   =   5'd2 ;
8'b00100111 :    rangeILps2   =   5'd2 ;
8'b00101000 :    rangeILps2   =   5'd2 ;
8'b00101001 :    rangeILps2   =   5'd2 ;
8'b00101010 :    rangeILps2   =   5'd2 ;
8'b00101011 :    rangeILps2   =   5'd2 ;
8'b00101100 :    rangeILps2   =   5'd2 ;
8'b00101101 :    rangeILps2   =   5'd2 ;
8'b00101110 :    rangeILps2   =   5'd2 ;
8'b00101111 :    rangeILps2   =   5'd2 ;
8'b00110000 :    rangeILps2   =   5'd2 ;
8'b00110001 :    rangeILps2   =   5'd2 ;
8'b00110010 :    rangeILps2   =   5'd2 ;
8'b00110011 :    rangeILps2   =   5'd2 ;
8'b00110100 :    rangeILps2   =   5'd2 ;
8'b00110101 :    rangeILps2   =   5'd2 ;
8'b00110110 :    rangeILps2   =   5'd2 ;
8'b00110111 :    rangeILps2   =   5'd2 ;
8'b00111000 :    rangeILps2   =   5'd2 ;
8'b00111001 :    rangeILps2   =   5'd2 ;
8'b00111010 :    rangeILps2   =   5'd2 ;
8'b00111011 :    rangeILps2   =   5'd2 ;
8'b00111100 :    rangeILps2   =   5'd2 ;
8'b00111101 :    rangeILps2   =   5'd2 ;
8'b00111110 :    rangeILps2   =   5'd2 ;
8'b00111111 :    rangeILps2   =   5'd2 ;
8'b01000000 :    rangeILps2   =   5'd1 ;
8'b01000001 :    rangeILps2   =   5'd1 ;
8'b01000010 :    rangeILps2   =   5'd1 ;
8'b01000011 :    rangeILps2   =   5'd1 ;
8'b01000100 :    rangeILps2   =   5'd1 ;
8'b01000101 :    rangeILps2   =   5'd1 ;
8'b01000110 :    rangeILps2   =   5'd1 ;
8'b01000111 :    rangeILps2   =   5'd1 ;
8'b01001000 :    rangeILps2   =   5'd1 ;
8'b01001001 :    rangeILps2   =   5'd1 ;
8'b01001010 :    rangeILps2   =   5'd1 ;
8'b01001011 :    rangeILps2   =   5'd1 ;
8'b01001100 :    rangeILps2   =   5'd1 ;
8'b01001101 :    rangeILps2   =   5'd1 ;
```

```
8'b01001110 :    rangeILps2   =   5'd1 ;
8'b01001111 :    rangeILps2   =   5'd1 ;
8'b01010000 :    rangeILps2   =   5'd1 ;
8'b01010001 :    rangeILps2   =   5'd1 ;
8'b01010010 :    rangeILps2   =   5'd1 ;
8'b01010011 :    rangeILps2   =   5'd1 ;
8'b01010100 :    rangeILps2   =   5'd1 ;
8'b01010101 :    rangeILps2   =   5'd1 ;
8'b01010110 :    rangeILps2   =   5'd1 ;
8'b01010111 :    rangeILps2   =   5'd1 ;
8'b01011000 :    rangeILps2   =   5'd1 ;
8'b01011001 :    rangeILps2   =   5'd1 ;
8'b01011010 :    rangeILps2   =   5'd1 ;
8'b01011011 :    rangeILps2   =   5'd1 ;
8'b01011100 :    rangeILps2   =   5'd1 ;
8'b01011101 :    rangeILps2   =   5'd1 ;
8'b01011110 :    rangeILps2   =   5'd1 ;
8'b01011111 :    rangeILps2   =   5'd1 ;
8'b01100000 :    rangeILps2   =   5'd1 ;
8'b01100001 :    rangeILps2   =   5'd1 ;
8'b01100010 :    rangeILps2   =   5'd1 ;
8'b01100011 :    rangeILps2   =   5'd1 ;
8'b01100100 :    rangeILps2   =   5'd1 ;
8'b01100101 :    rangeILps2   =   5'd1 ;
8'b01100110 :    rangeILps2   =   5'd1 ;
8'b01100111 :    rangeILps2   =   5'd1 ;
8'b01101000 :    rangeILps2   =   5'd1 ;
8'b01101001 :    rangeILps2   =   5'd1 ;
8'b01101010 :    rangeILps2   =   5'd1 ;
8'b01101011 :    rangeILps2   =   5'd1 ;
8'b01101100 :    rangeILps2   =   5'd1 ;
8'b01101101 :    rangeILps2   =   5'd1 ;
8'b01101110 :    rangeILps2   =   5'd1 ;
8'b01101111 :    rangeILps2   =   5'd1 ;
8'b01110000 :    rangeILps2   =   5'd1 ;
8'b01110001 :    rangeILps2   =   5'd1 ;
8'b01110010 :    rangeILps2   =   5'd1 ;
8'b01110011 :    rangeILps2   =   5'd1 ;
8'b01110100 :    rangeILps2   =   5'd1 ;
8'b01110101 :    rangeILps2   =   5'd1 ;
8'b01110110 :    rangeILps2   =   5'd1 ;
8'b01110111 :    rangeILps2   =   5'd1 ;
8'b01111000 :    rangeILps2   =   5'd1 ;
8'b01111001 :    rangeILps2   =   5'd1 ;
```

```
            8'b01111010 :    rangeILps2   =   5'd1 ;
            8'b01111011 :    rangeILps2   =   5'd1 ;
            8'b01111100 :    rangeILps2   =   5'd1 ;
            8'b01111101 :    rangeILps2   =   5'd1 ;
            8'b01111110 :    rangeILps2   =   5'd1 ;
            8'b01111111 :    rangeILps2   =   5'd1 ;
            default   :    rangeILps2   =   5'd0 ;
            endcase
        end

endmodule
```

A.1.2 Offset Update Module(dOffsetUpdate.v)

```
`timescale 1ns/100ps

module dOffsetUpdate (
    input                           clk,
    input                           rst_n,

    // Signals from Context Modeling
    input                           i_reset,
    input                           i_init,
    input                           i_valid,

    // Signals for Updating Offset
    input                           i_isLPS,
    input       [7:0]               i_rangeFMps,
    input       [4:0]               i_rangeILps,

    // Signals from ReadBit
    input                           i_readBits1,
    input       [7:0]               i_readBits2,
    input       [31:0]              i_readBits3,

    // Signals to ReadBit
    output                          o_numOfReadBits1,
    output      [3:0]               o_numOfReadBits2,
    output      [4:0]               o_numOfReadBits3,
```

```
    // Signals to Range Update
    output                          o_valid,
    output      [7:0]               o_offsetF,
    output      [4:0]               o_offsetI,

    // Signals for Test  ,
    output      [7:0]               t_offsetF,
    output      [4:0]               t_offsetI
);

//////////////////////////////////////////////////////////////////////////////
// offset Update Stage
//////////////////////////////////////////////////////////////////////////////
    reg         [7:0]               reg_offsetF         ;
    reg         [4:0]               reg_offsetI         ;
    wire    [7:0]                   updated_offsetF     ;
    wire    [4:0]                   updated_offsetI     ;

    wire                            s_flag_offset   ;

    wire    [8:0]                   un_offsetF          ;
    wire    [9:0]                   us_offsetF          ;
    wire    [8:0]                   u_offsetF           ;
    wire                            u_offsetI       ;

    reg         [8:0]               s_offsetF       ;

    reg         [7:0]               n_offsetF           ;
    reg         [4:0]               n_offsetI       ;

    //////////////////////////////////////////////////////////////////////////
    // Output
    //////////////////////////////////////////////////////////////////////////
    assign o_valid                  = i_valid ;
    assign o_offsetF            = reg_offsetF ;
    assign o_offsetI            = reg_offsetI ;
    assign t_offsetF            = updated_offsetF ;
    assign t_offsetI            = updated_offsetI ;

    assign o_numOfReadBits1     = i_init | s_flag_offset ;
    assign o_numOfReadBits2 = ( i_init == 1'b1 ) ? 4'd8 : i_rangeILps[3:0] ;
    assign o_numOfReadBits3 = n_offsetI ;

    //////////////////////////////////////////////////////////////////////////
```

```
    // s_flag
    ////////////////////////////////////////////////////////////////////////////
    assign s_flag_offset        = ( i_init == 1'b1 || reg_offsetF < i_rangeFMps ) ? 1'b1 :
1'b0 ;

    ////////////////////////////////////////////////////////////////////////////
    // OffsetF Update
    ////////////////////////////////////////////////////////////////////////////
    assign un_offsetF           = {1'b0,reg_offsetF} - {1'b0,i_rangeFMps} ;
        // non scaled offsetF
    assign us_offsetF           =  10'd256  +  {1'b0,reg_offsetF[7:0],i_readBits1}  -
{2'b0,i_rangeFMps} ;        // scaled offsetF
    assign u_offsetF            = ( s_flag_offset == 1'b1 ) ? us_offsetF[8:0] : un_offsetF ;

    ////////////////////////////////////////////////////////////////////////////
    // OffsetI Update
    ////////////////////////////////////////////////////////////////////////////
    assign u_offsetI            = ( s_flag_offset == 1'b1 ) ? 1'b1 : 1'b0 ;

    ////////////////////////////////////////////////////////////////////////////
    // rangeF Scaling ( renormalization )
    ////////////////////////////////////////////////////////////////////////////
    always@(i_rangeILps,u_offsetF,i_readBits2,i_init) begin
        if ( i_init == 1'b1 ) begin
            s_offsetF               = {u_offsetF[0:0],i_readBits2[7:0]} ;
        end else begin
            case(i_rangeILps)
            4'd1 :    s_offsetF = {u_offsetF[7:0],i_readBits2[7:7]} ;
            4'd2 :    s_offsetF = {u_offsetF[6:0],i_readBits2[7:6]} ;
            4'd3 :    s_offsetF = {u_offsetF[5:0],i_readBits2[7:5]} ;
            4'd4 :    s_offsetF = {u_offsetF[4:0],i_readBits2[7:4]} ;
            4'd5 :    s_offsetF = {u_offsetF[3:0],i_readBits2[7:3]} ;
            4'd6 :    s_offsetF = {u_offsetF[2:0],i_readBits2[7:2]} ;
            4'd7 :    s_offsetF = {u_offsetF[1:0],i_readBits2[7:1]} ;
            4'd8 :    s_offsetF = {u_offsetF[0:0],i_readBits2[7:0]} ;
            default :    s_offsetF = u_offsetF ;
            endcase
        end
    end

    ////////////////////////////////////////////////////////////////////////////
    // offsetF Scaling ( domain conversion )
    ////////////////////////////////////////////////////////////////////////////
    wire [40:0]         e_offsetF ;
```

```
assign e_offsetF = {s_offsetF,i_readBits3} ;

always@(e_offsetF) begin
    if ( e_offsetF[40:40] == 1 ) begin
        n_offsetF  = e_offsetF[39:32]  ;
        n_offsetI = 5'd0               ;
    end else if ( e_offsetF[40:39] == 1 ) begin
        n_offsetF  = e_offsetF[38:31]  ;
        n_offsetI = 5'd1               ;
    end else if ( e_offsetF[40:38] == 1 ) begin
        n_offsetF  = e_offsetF[37:30]  ;
        n_offsetI = 5'd2               ;
    end else if ( e_offsetF[40:37] == 1 ) begin
        n_offsetF  = e_offsetF[36:29]  ;
        n_offsetI = 5'd3               ;
    end else if ( e_offsetF[40:36] == 1 ) begin
        n_offsetF  = e_offsetF[35:28]  ;
        n_offsetI = 5'd4               ;
    end else if ( e_offsetF[40:35] == 1 ) begin
        n_offsetF  = e_offsetF[34:27]  ;
        n_offsetI = 5'd5               ;
    end else if ( e_offsetF[40:34] == 1 ) begin
        n_offsetF  = e_offsetF[33:26]  ;
        n_offsetI = 5'd6               ;
    end else if ( e_offsetF[40:33] == 1 ) begin
        n_offsetF  = e_offsetF[32:25]  ;
        n_offsetI = 5'd7               ;
    end else if ( e_offsetF[40:32] == 1 ) begin
        n_offsetF  = e_offsetF[31:24]  ;
        n_offsetI = 5'd8               ;
    end else if ( e_offsetF[40:31] == 1 ) begin
        n_offsetF  = e_offsetF[30:23]  ;
        n_offsetI = 5'd9               ;
    end else if ( e_offsetF[40:30] == 1 ) begin
        n_offsetF  = e_offsetF[29:22]  ;
        n_offsetI = 5'd10              ;
    end else if ( e_offsetF[40:29] == 1 ) begin
        n_offsetF  = e_offsetF[28:21]  ;
        n_offsetI = 5'd11              ;
    end else if ( e_offsetF[40:28] == 1 ) begin
        n_offsetF  = e_offsetF[27:20]  ;
        n_offsetI = 5'd12              ;
    end else if ( e_offsetF[40:27] == 1 ) begin
```

```
            n_offsetF       = e_offsetF[26:19]      ;
            n_offsetI = 5'd13                       ;
        end else if ( e_offsetF[40:26] == 1 ) begin
            n_offsetF       = e_offsetF[25:18]      ;
            n_offsetI = 5'd14                       ;
        end else if ( e_offsetF[40:25] == 1 ) begin
            n_offsetF       = e_offsetF[24:17]      ;
            n_offsetI = 5'd15                       ;
        end else if ( e_offsetF[40:24] == 1 ) begin
            n_offsetF       = e_offsetF[23:16]      ;
            n_offsetI = 5'd16                       ;
        end else if ( e_offsetF[40:23] == 1 ) begin
            n_offsetF       = e_offsetF[22:15]      ;
            n_offsetI = 5'd17                       ;
        end else if ( e_offsetF[40:22] == 1 ) begin
            n_offsetF       = e_offsetF[21:14]      ;
            n_offsetI = 5'd18                       ;
        end else begin
            n_offsetF       = e_offsetF[20:13]      ;
            n_offsetI = 5'd19                       ;
        end
    end

    ////////////////////////////////////////////////////////////////////////////
    // offsetF Update
    ////////////////////////////////////////////////////////////////////////////
    assign updated_offsetF      = ( i_isLPS ) ? n_offsetF : reg_offsetF ;

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
            reg_offsetF             <= 8'd0 ;
        end else begin
            if ( i_reset == 1'b1 ) begin
                reg_offsetF         <= 8'd0 ;
            end else if ( i_valid == 1'b1 || i_init == 1'b1 ) begin
                reg_offsetF         <= updated_offsetF ;
            end
        end
    end

    ////////////////////////////////////////////////////////////////////////////
    // offsetI Update
    ////////////////////////////////////////////////////////////////////////////
    assign updated_offsetI      = ( i_isLPS ) ? n_offsetI : reg_offsetI ;
```

```
    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
            reg_offsetI             <= 5'b0 ;
        end else begin
            if ( i_reset == 1'b1 ) begin
                reg_offsetI         <= 5'b0 ;
            end else if ( i_valid == 1'b1 || i_init == 1'b1 ) begin
                reg_offsetI         <= updated_offsetI ;
            end
        end
    end

endmodule
```

A.1.3 Bits Read Module (dReadBits.v)

```
`timescale 1ns/100ps

module dReadBits #(
    parameter ADDR_WIDTH                = 16
)(
    input                               clk,
    input                               rst_n,

    input                               i_init,
    input                               i_valid,
    input                               i_isLPS,

    // form Bitstream Buffer
    output                              renable,
    output  reg [ADDR_WIDTH-1:0]        raddr,
    input       [63:0]                  rdata,

    input                               i_numOfReadBits1,
    input       [3:0]                   i_numOfReadBits2,
    input       [4:0]                   i_numOfReadBits3,

    output                              o_readBits1,
    output      [7:0]                   o_readBits2,
    output      [31:0]                  o_readBits3
```

```
);
    reg         [5:0]           currBitCount    ;
    wire    [6:0]           nextBitCount1   ;
    wire    [6:0]           nextBitCount2   ;
    wire    [6:0]           nextBitCount3   ;
    wire    [6:0]           nextBitCount4   ;

    reg         [63:0]          currBitBuffer   ;
    reg         [63:0]          nextBitBuffer1      ;
    reg         [63:0]          nextBitBuffer2      ;
    reg         [63:0]          nextBitBuffer3      ;
    wire    [63:0]          nextBitBuffer4      ;

    reg         [63:0]          currPreLoadBuffer;
    reg         [63:0]          nextPreLoadBuffer0  ;
    reg         [63:0]          nextPreLoadBuffer1  ;
    reg         [63:0]          nextPreLoadBuffer2  ;
    reg         [63:0]          nextPreLoadBuffer3  ;
    wire    [63:0]          nextPreLoadBuffer4  ;

    reg                         init_1d             ;
    reg                         renable_1d          ;

    assign o_readBits1 = currBitBuffer[63] ;
    assign o_readBits2 = nextBitBuffer1[63:56] ;
    assign o_readBits3 = nextBitBuffer2[63:32] ;

    always@(currBitCount,currPreLoadBuffer,rdata) begin
        case(currBitCount[5:0])
        6'd63   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:63],rdata[63:01]} ;
        6'd62   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:62],rdata[63:02]} ;
        6'd61   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:61],rdata[63:03]} ;
        6'd60   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:60],rdata[63:04]} ;
        6'd59   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:59],rdata[63:05]} ;
        6'd58   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:58],rdata[63:06]} ;
        6'd57   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:57],rdata[63:07]} ;
        6'd56   :   nextPreLoadBuffer0  =
```

```
{currPreLoadBuffer[63:56],rdata[63:08]} ;
        6'd55   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:55],rdata[63:09]} ;
        6'd54   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:54],rdata[63:10]} ;
        6'd53   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:53],rdata[63:11]} ;
        6'd52   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:52],rdata[63:12]} ;
        6'd51   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:51],rdata[63:13]} ;
        6'd50   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:50],rdata[63:14]} ;
        6'd49   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:49],rdata[63:15]} ;
        6'd48   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:48],rdata[63:16]} ;
        6'd47   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:47],rdata[63:17]} ;
        6'd46   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:46],rdata[63:18]} ;
        6'd45   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:45],rdata[63:19]} ;
        6'd44   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:44],rdata[63:20]} ;
        6'd43   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:43],rdata[63:21]} ;
        6'd42   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:42],rdata[63:22]} ;
        6'd41   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:41],rdata[63:23]} ;
        6'd40   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:40],rdata[63:24]} ;
        6'd39   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:39],rdata[63:25]} ;
        6'd38   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:38],rdata[63:26]} ;
        6'd37   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:37],rdata[63:27]} ;
        6'd36   :   nextPreLoadBuffer0  =
{currPreLoadBuffer[63:36],rdata[63:28]} ;
        default :   nextPreLoadBuffer0  = currPreLoadBuffer ;
        endcase
    end
```

```
    assign nextBitCount1 = {1'b0,currBitCount} + {6'b0,i_numOfReadBits1} ;

    always@(i_numOfReadBits1,currBitBuffer,nextPreLoadBuffer0) begin
        if ( i_numOfReadBits1 == 1'b1 ) begin
            nextBitBuffer1              =
{currBitBuffer[62:0],nextPreLoadBuffer0[63]} ;
            nextPreLoadBuffer1          = {nextPreLoadBuffer0[62:0],1'b0} ;
        end else begin
            nextBitBuffer1              = currBitBuffer ;
            nextPreLoadBuffer1          = nextPreLoadBuffer0 ;
        end
    end

    assign nextBitCount2 = nextBitCount1 + {3'b0,i_numOfReadBits2} ;

    always@(i_numOfReadBits2,nextBitBuffer1,nextPreLoadBuffer1) begin
        case(i_numOfReadBits2)
        4'd1 :    nextBitBuffer2    =
{nextBitBuffer1[62:0],nextPreLoadBuffer1[63:63]} ;
        4'd2 :    nextBitBuffer2    =
{nextBitBuffer1[61:0],nextPreLoadBuffer1[63:62]} ;
        4'd3 :    nextBitBuffer2    =
{nextBitBuffer1[60:0],nextPreLoadBuffer1[63:61]} ;
        4'd4 :    nextBitBuffer2    =
{nextBitBuffer1[59:0],nextPreLoadBuffer1[63:60]} ;
        4'd5 :    nextBitBuffer2    =
{nextBitBuffer1[58:0],nextPreLoadBuffer1[63:59]} ;
        4'd6 :    nextBitBuffer2    =
{nextBitBuffer1[57:0],nextPreLoadBuffer1[63:58]} ;
        4'd7 :    nextBitBuffer2    =
{nextBitBuffer1[56:0],nextPreLoadBuffer1[63:57]} ;
        4'd8 :    nextBitBuffer2    =
{nextBitBuffer1[55:0],nextPreLoadBuffer1[63:56]} ;
        default :    nextBitBuffer2    = nextBitBuffer1 ;
        endcase
    end

    always@(i_numOfReadBits2,nextPreLoadBuffer1) begin
        case(i_numOfReadBits2)
        4'd1 :    nextPreLoadBuffer2    = {nextPreLoadBuffer1[62:0],1'b0} ;
        4'd2 :    nextPreLoadBuffer2    = {nextPreLoadBuffer1[61:0],2'b0} ;
        4'd3 :    nextPreLoadBuffer2    = {nextPreLoadBuffer1[60:0],3'b0} ;
        4'd4 :    nextPreLoadBuffer2    = {nextPreLoadBuffer1[59:0],4'b0} ;
```

```
            4'd5 :      nextPreLoadBuffer2     = {nextPreLoadBuffer1[58:0],5'b0} ;
            4'd6 :      nextPreLoadBuffer2     = {nextPreLoadBuffer1[57:0],6'b0} ;
            4'd7 :      nextPreLoadBuffer2     = {nextPreLoadBuffer1[56:0],7'b0} ;
            4'd8 :      nextPreLoadBuffer2     = {nextPreLoadBuffer1[55:0],8'b0} ;
            default   :      nextPreLoadBuffer2     = nextPreLoadBuffer1 ;
            endcase
        end

        assign nextBitCount3 = nextBitCount2 + {2'b0,i_numOfReadBits3} ;

        always@(i_numOfReadBits3,nextBitBuffer2,nextPreLoadBuffer2) begin
            case(i_numOfReadBits3)
            5'd1 :      nextBitBuffer3     =
{nextBitBuffer2[62:0],nextPreLoadBuffer2[63:63]} ;
            5'd2 :      nextBitBuffer3     =
{nextBitBuffer2[61:0],nextPreLoadBuffer2[63:62]} ;
            5'd3 :      nextBitBuffer3     =
{nextBitBuffer2[60:0],nextPreLoadBuffer2[63:61]} ;
            5'd4 :      nextBitBuffer3     =
{nextBitBuffer2[59:0],nextPreLoadBuffer2[63:60]} ;
            5'd5 :      nextBitBuffer3     =
{nextBitBuffer2[58:0],nextPreLoadBuffer2[63:59]} ;
            5'd6 :      nextBitBuffer3     =
{nextBitBuffer2[57:0],nextPreLoadBuffer2[63:58]} ;
            5'd7 :      nextBitBuffer3     =
{nextBitBuffer2[56:0],nextPreLoadBuffer2[63:57]} ;
            5'd8 :      nextBitBuffer3     =
{nextBitBuffer2[55:0],nextPreLoadBuffer2[63:56]} ;
            5'd9 :      nextBitBuffer3     =
{nextBitBuffer2[54:0],nextPreLoadBuffer2[63:55]} ;
            5'd10     :     nextBitBuffer3     =
{nextBitBuffer2[53:0],nextPreLoadBuffer2[63:54]} ;
            5'd11     :     nextBitBuffer3     =
{nextBitBuffer2[52:0],nextPreLoadBuffer2[63:53]} ;
            5'd12     :     nextBitBuffer3     =
{nextBitBuffer2[51:0],nextPreLoadBuffer2[63:52]} ;
            5'd13     :     nextBitBuffer3     =
{nextBitBuffer2[50:0],nextPreLoadBuffer2[63:51]} ;
            5'd14     :     nextBitBuffer3     =
{nextBitBuffer2[49:0],nextPreLoadBuffer2[63:50]} ;
            5'd15     :     nextBitBuffer3     =
{nextBitBuffer2[48:0],nextPreLoadBuffer2[63:49]} ;
            5'd16     :     nextBitBuffer3     =
{nextBitBuffer2[47:0],nextPreLoadBuffer2[63:48]} ;
```

```
            5'd17    :    nextBitBuffer3    =
{nextBitBuffer2[46:0],nextPreLoadBuffer2[63:47]} ;
            5'd18    :    nextBitBuffer3    =
{nextBitBuffer2[45:0],nextPreLoadBuffer2[63:46]} ;
            5'd19    :    nextBitBuffer3    =
{nextBitBuffer2[44:0],nextPreLoadBuffer2[63:45]} ;
            default  :    nextBitBuffer3    = nextBitBuffer2 ;
            endcase
        end

        always@(i_numOfReadBits3,nextPreLoadBuffer2) begin
            case(i_numOfReadBits3)
            5'd1 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[62:0],01'b0} ;
            5'd2 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[61:0],02'b0} ;
            5'd3 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[60:0],03'b0} ;
            5'd4 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[59:0],04'b0} ;
            5'd5 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[58:0],05'b0} ;
            5'd6 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[57:0],06'b0} ;
            5'd7 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[56:0],07'b0} ;
            5'd8 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[55:0],08'b0} ;
            5'd9 :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[54:0],09'b0} ;
            5'd10    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[53:0],10'b0} ;
            5'd11    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[52:0],11'b0} ;
            5'd12    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[51:0],12'b0} ;
            5'd13    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[50:0],13'b0} ;
            5'd14    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[49:0],14'b0} ;
            5'd15    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[48:0],15'b0} ;
            5'd16    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[47:0],16'b0} ;
            5'd17    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[46:0],17'b0} ;
            5'd18    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[45:0],18'b0} ;
            5'd19    :    nextPreLoadBuffer3    = {nextPreLoadBuffer2[44:0],19'b0} ;
            default  :    nextPreLoadBuffer3    = nextPreLoadBuffer2 ;
            endcase
        end

        assign nextBitCount4        = ( i_isLPS == 1'b1 ) ? nextBitCount3          :
currBitCount      ;
        assign nextBitBuffer4       = ( i_isLPS == 1'b1 ) ? nextBitBuffer3         :
currBitBuffer     ;
        assign nextPreLoadBuffer4   = ( i_isLPS == 1'b1 ) ? nextPreLoadBuffer3     :
currPreLoadBuffer ;

        always@(posedge clk,negedge rst_n) begin
            if (!rst_n) begin
```

```
            currBitCount        <= 6'd0 ;
        end else begin
            if ( i_init == 1'b1 ) begin
                currBitCount  <= 6'd0 ;
            end else if ( i_valid == 1'b1 ) begin
                currBitCount  <= nextBitCount4[5:0] ;
            end
        end
    end

    always@(posedge clk) begin
        if ( init_1d == 1'b1 ) begin
            currBitBuffer       <= currPreLoadBuffer ;
        end else if ( i_valid == 1'b1 ) begin
            currBitBuffer       <= nextBitBuffer4 ;
        end
    end

    always@(posedge clk) begin
        if ( init_1d == 1'b1 ) begin
            currPreLoadBuffer<= rdata ;
        end else if ( renable == 1'b1 ) begin
            case(nextBitCount4[5:0])
            6'd01    :    currPreLoadBuffer<= {rdata[62:0],01'b0} ;
            6'd02    :    currPreLoadBuffer<= {rdata[61:0],02'b0} ;
            6'd03    :    currPreLoadBuffer<= {rdata[60:0],03'b0} ;
            6'd04    :    currPreLoadBuffer<= {rdata[59:0],04'b0} ;
            6'd05    :    currPreLoadBuffer<= {rdata[58:0],05'b0} ;
            6'd06    :    currPreLoadBuffer<= {rdata[57:0],06'b0} ;
            6'd07    :    currPreLoadBuffer<= {rdata[56:0],07'b0} ;
            6'd08    :    currPreLoadBuffer<= {rdata[55:0],08'b0} ;
            6'd09    :    currPreLoadBuffer<= {rdata[54:0],09'b0} ;
            6'd10    :    currPreLoadBuffer<= {rdata[53:0],10'b0} ;
            6'd11    :    currPreLoadBuffer<= {rdata[52:0],11'b0} ;
            6'd12    :    currPreLoadBuffer<= {rdata[51:0],12'b0} ;
            6'd13    :    currPreLoadBuffer<= {rdata[50:0],13'b0} ;
            6'd14    :    currPreLoadBuffer<= {rdata[49:0],14'b0} ;
            6'd15    :    currPreLoadBuffer<= {rdata[48:0],15'b0} ;
            6'd16    :    currPreLoadBuffer<= {rdata[47:0],16'b0} ;
            6'd17    :    currPreLoadBuffer<= {rdata[46:0],17'b0} ;
            6'd18    :    currPreLoadBuffer<= {rdata[45:0],18'b0} ;
            6'd19    :    currPreLoadBuffer<= {rdata[44:0],19'b0} ;
            6'd20    :    currPreLoadBuffer<= {rdata[43:0],20'b0} ;
            6'd21    :    currPreLoadBuffer<= {rdata[42:0],21'b0} ;
```

```
                6'd22    :    currPreLoadBuffer<= {rdata[41:0],22'b0} ;
                6'd23    :    currPreLoadBuffer<= {rdata[40:0],23'b0} ;
                6'd24    :    currPreLoadBuffer<= {rdata[39:0],24'b0} ;
                6'd25    :    currPreLoadBuffer<= {rdata[38:0],25'b0} ;
                6'd26    :    currPreLoadBuffer<= {rdata[37:0],26'b0} ;
                6'd27    :    currPreLoadBuffer<= {rdata[36:0],27'b0} ;
                6'd28    :    currPreLoadBuffer<= {rdata[35:0],28'b0} ;
                6'd29    :    currPreLoadBuffer<= {rdata[34:0],29'b0} ;
                default :  currPreLoadBuffer<= rdata ;
                endcase
        end else if ( i_valid == 1'b1 ) begin
                currPreLoadBuffer<= nextPreLoadBuffer4 ;
        end
    end

    assign renable = nextBitCount4[6]&i_valid | i_init ;

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
                init_1d                 <= 1'b0 ;
        end else begin
                init_1d                 <= i_init ;
        end
    end

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
                renable_1d              <= 1'b0 ;
        end else begin
                renable_1d              <= nextBitCount4[6]&i_valid ;
        end
    end

    always@(posedge clk,negedge rst_n) begin
        if (!rst_n) begin
                raddr                   <= {ADDR_WIDTH{1'b0}} ;
        end else begin
                if ( renable == 1'b1 ) begin
                    raddr               <= raddr + 1 ;
                end
        end
    end

endmodule
```

A.1.4 Binary Arithmetic Decoding Top Module (BADTop.v)

```
`timescale 1ns/100ps

module BADTop #(
    parameter ADDR_WIDTH                = 16
)(
    input                               clk,
    input                               rst_n,

    // Signals from Context Modeling
    input                               i_reset,
    input                               i_init_offset,
    input                               i_init_readBits,
    input                               i_valid,
    input                               i_valMPS,
    input       [7:0]                   i_lgPmps,

    // form Bitstream Buffer
    output                              renable,
    output      [ADDR_WIDTH-1:0]        raddr,
    input       [63:0]                  rdata,

    // Signals to Context Modeling
    output                              o_valid,
    output                              o_decodedBin,

    // Signals for Test ,
    output                              t_isLPS,
    output      [7:0]                   t_rangeF,
    output      [4:0]                   t_rangeI,
    output      [7:0]                   t_offsetF,
    output      [4:0]                   t_offsetI
);
    wire    [7:0]               offsetF         ;
    wire    [4:0]               offsetI         ;
    wire                        isLPS           ;
    wire        [7:0]           rangeFMps       ;
    wire        [4:0]           rangeILps       ;

    wire                        readBits1       ;
```

```
wire    [7:0]           readBits2           ;
wire    [31:0]          readBits3           ;
wire                    numOfReadBits1      ;
wire    [3:0]           numOfReadBits2      ;
wire    [4:0]           numOfReadBits3      ;

assign t_isLPS = isLPS ;

dRangeUpdate A_dRangeUpdate(
    .clk            ( clk               ),
    .rst_n          ( rst_n             ),
    .i_reset        ( i_reset           ),
    .i_valid        ( i_valid           ),
    .i_valMPS       ( i_valMPS          ),
    .i_lgPmps       ( i_lgPmps          ),
    .i_offsetI      ( offsetI           ),
    .i_offsetF      ( offsetF           ),
    .o_valid        ( o_valid           ),
    .o_decodedBin   ( o_decodedBin      ),
    .o_isLPS        ( isLPS             ),
    .o_rangeFMps    ( rangeFMps         ),
    .o_rangeILps    ( rangeILps         ),
    .t_rangeF       ( t_rangeF          ),
    .t_rangeI       ( t_rangeI          )
);

dOffsetUpdate A_dOffsetUpdate(
    .clk            ( clk               ),
    .rst_n          ( rst_n             ),
    .i_reset        ( i_reset           ),
    .i_init         ( i_init_offset     ),
    .i_valid        ( i_valid           ),
    .i_isLPS        ( isLPS | i_init_offset ),
    .i_rangeFMps    ( rangeFMps         ),
    .i_rangeILps    ( rangeILps         ),
    .i_readBits1    ( readBits1         ),
    .i_readBits2    ( readBits2         ),
    .i_readBits3    ( readBits3         ),
    .o_numOfReadBits1   ( numOfReadBits1    ),
    .o_numOfReadBits2   ( numOfReadBits2    ),
    .o_numOfReadBits3   ( numOfReadBits3    ),
    .o_valid        ( /*open*/          ),
    .o_offsetF      ( offsetF           ),
    .o_offsetI      ( offsetI           ),
```

```
        .t_offsetF          ( t_offsetF          ),
        .t_offsetI          ( t_offsetI          )
    );

    dReadBits #(ADDR_WIDTH) A_dReadBits(
        .clk                ( clk                ),
        .rst_n              ( rst_n              ),
        .i_init             ( i_init_readBits    ),
        .i_valid            ( i_valid | i_init_offset),
        .i_isLPS            ( isLPS | i_init_offset ),
        .renable            ( renable            ),
        .raddr              ( raddr              ),
        .rdata              ( rdata              ),
        .i_numOfReadBits1   ( numOfReadBits1     ),
        .i_numOfReadBits2   ( numOfReadBits2     ),
        .i_numOfReadBits3   ( numOfReadBits3     ),
        .o_readBits1        ( readBits1          ),
        .o_readBits2        ( readBits2          ),
        .o_readBits3        ( readBits3          )
    );

endmodule
```

A.1.5 Test Bench

```
`timescale 1ns/100ps

module tb ( ) ;
    reg clk    ;
    reg rst_n ;
    initial begin
        clk  = 0 ;
        rst_n    = 0 ;
        #10
        rst_n    = 1 ;
    end

    always begin
        #2.5 clk  <= ~clk ;
    end
```

```
    dTB_Single_Bin A_dTB_Single_Bin (clk,rst_n) ;

endmodule
```

```
`timescale 1ns/100ps

module dTB_Single_Bin (
    input                               clk,        // Clock input
    input                               rst_n       // Reset async input active low
);
    import "DPI-C" context task dMain_single_bin();
    export "DPI-C" task dTb_single_bin_wait_clk;
    export "DPI-C" task dTb_single_bin_wait_rstn;
    export "DPI-C" task dTb_single_bin_input_write;
    export "DPI-C" task dTb_single_bin_output_read;
    export "DPI-C" task dTb_single_bin_init;
    export "DPI-C" task dTb_single_bin_writeBitStream;

    reg                                 iReset      ;
    reg                                 init_offset ;
    reg                                 init_readBits ;
    reg                                 iValid      ;
    wire                                oValid      ;
    reg                                 oClear      ;
    reg                                 output_valid;

    reg     [31:0]                      iBinCount   ;
    reg                                 iValMPS     ;
    reg     [7:0]                       iLgPmps     ;
    reg     [4:0]                       iOffsetI    ;
    reg     [7:0]                       iOffsetF    ;

    reg     [7:0]                       oRangeF     ;
    reg     [4:0]                       oRangeI     ;
    reg                                 oIsLPS      ;
    reg                                 oDecodedBin ;
    reg     [4:0]                       oOffsetI    ;
    reg     [7:0]                       oOffsetF    ;

    wire[7:0]                       reg_RangeF      ;
    wire[4:0]                       reg_RangeI      ;
    wire                            reg_IsLPS       ;
    wire                            reg_DecodedBin  ;
```

```
    wire[7:0]                       reg_OffsetF     ;
    wire[4:0]                       reg_OffsetI     ;

    parameter ADDR_WIDTH                = 16;

    reg                                 wenable     ;
    reg     [ADDR_WIDTH-1:0]            waddr       ;
    reg     [63:0]                      wdata       ;
    wire                            renable     ;
    wire[ADDR_WIDTH-1:0]            raddr       ;
    wire[63:0]                      rdata       ;

    initial begin
        iValid              = 1'b0 ;
        iReset              = 1'b0 ;
        init_offset         = 1'b0 ;
        init_readBits       = 1'b0 ;
    end

    initial begin
        repeat(30) @(posedge clk);
        dMain_single_bin();
    end

    task dTb_single_bin_wait_clk (input int cycle);
        repeat(cycle) @(posedge clk);
    endtask

    task dTb_single_bin_wait_rstn (output bit o_rst_n);
        while(!rst_n) begin
            @(posedge clk);
        end
        o_rst_n = rst_n;
    endtask

    task dTb_single_bin_input_write (input int i_mode,input int i_binCount,input int
i_valMPS,input int i_lgPmps,input int i_offsetI,input int i_offsetF);
        iValid                  <= 1'b1 ;
        iBinCount               <= i_binCount ;
        iValMPS                 <= i_valMPS ;
        iLgPmps                 <= i_lgPmps[9:2] ;
        iOffsetI            <= i_offsetI ;
        iOffsetF            <= i_offsetF ;
        repeat(1) @(posedge clk);
```

```
        iValid                  <= 1'b0 ;
    endtask

    task dTb_single_bin_output_read (output int o_valid,output int o_rangeF,output int o_rangeI,output int o_offsetF,output int o_offsetI,output int o_isLPS,output int o_decodedBin);
        o_valid                 <= output_valid ;
        oClear                  <= 1'b1         ;
        o_rangeF            <= oRangeF      ;
        o_rangeI            <= oRangeI      ;
        o_offsetF               <= oOffsetF   ;
        o_offsetI           <= oOffsetI    ;
        o_isLPS                 <= oIsLPS         ;
        o_decodedBin            <= oDecodedBin   ;
        repeat(1) @(posedge clk);
        oClear                  <= 1'b0         ;
    endtask

    task dTb_single_bin_init(input int cycle);
        repeat(1) @(posedge clk);
        init_readBits           <= 1'b1 ;
        repeat(2) @(posedge clk);
        init_readBits           <= 1'b0 ;
        repeat(2) @(posedge clk);
        init_offset             <= 1'b1 ;
        repeat(1) @(posedge clk);
        init_offset             <= 1'b0 ;
        repeat(cycle) @(posedge clk);
    endtask

    task dTb_single_bin_writeBitStream(input int i_data[8]) ;
        wenable                 <= 1'b1 ;
        wdata[63:56]            <= i_data[0] ;
        wdata[55:48]            <= i_data[1] ;
        wdata[47:40]            <= i_data[2] ;
        wdata[39:32]            <= i_data[3] ;
        wdata[31:24]            <= i_data[4] ;
        wdata[23:16]            <= i_data[5] ;
        wdata[15:08]            <= i_data[6] ;
        wdata[07:00]            <= i_data[7] ;
        repeat(1) @(posedge clk);
        wenable                 <= 1'b0 ;
    endtask
```

```
always@(posedge clk,negedge rst_n) begin
    if (!rst_n) begin
        waddr                   <= {ADDR_WIDTH{1'b0}} ;
    end else if ( wenable == 1'b1 ) begin
        waddr                   <= waddr + 1 ;
    end
end

BADTop #(16) A_BADTop(
    .clk            ( clk               ),
    .rst_n          ( rst_n             ),
    .i_reset        ( iReset            ),
    .i_init_offset  ( init_offset       ),
    .i_init_readBits  ( init_readBits   ),
    .i_valid        ( iValid            ),
    .i_valMPS       ( iValMPS           ),
    .i_lgPmps       ( iLgPmps           ),
    .renable        ( renable           ),
    .raddr          ( raddr             ),
    .rdata          ( rdata             ),
    .o_valid        ( oValid            ),
    .o_decodedBin   ( reg_DecodedBin    ),
    .t_isLPS        ( reg_IsLPS         ),
    .t_rangeF       ( reg_RangeF        ),
    .t_rangeI       ( reg_RangeI        ),
    .t_offsetF      ( reg_OffsetF       ),
    .t_offsetI      ( reg_OffsetI       )
);

rf_memory #(64,ADDR_WIDTH) A_BitStreamBuffer(
    .clk            ( clk               ),
    .wenable        ( wenable           ),
    .waddr          ( waddr             ),
    .wdata          ( wdata             ),
    .renable        ( 1'b1              ),
    .raddr          ( raddr             ),
    .rdata          ( rdata             )
);

always@(posedge clk,negedge rst_n) begin
    if (!rst_n) begin
        output_valid   <= 1'b0 ;
    end else if ( oValid == 1'b1 ) begin
        output_valid       <= 1'b1 ;
```

```
        end else if ( oClear == 1'b1 ) begin
            output_valid      <= 1'b0 ;
        end
    end

    always@(posedge clk) begin
        if ( oValid == 1'b1 ) begin
            oRangeF       <= reg_RangeF          ;
            oRangeI       <= reg_RangeI          ;
            oOffsetF <= reg_OffsetF          ;
            oOffsetI  <= reg_OffsetI         ;
            oIsLPS        <= reg_IsLPS       ;
            oDecodedBin <= reg_DecodedBin    ;
        end
    end

endmodule
```